MPTCP
Performances et optimisation de la sécurité avec un ordonnancement réparti dans les topologies virtualisées OpenFlow

Encadrants : **S. Secci,**
Y. Benchaïb, M. Coudron,

Etudiants : **R. Ly, K. Lam, Q. Dubois, S. Ravier**

Table des matières

1 Cahier des charges **3**
1.1 Objectifs . . . 3
1.2 Contexte . . . 3
1.3 Méthodes . . . 3

2 Plan de développement **5**
2.1 Scripts de tests mininet . . . 6
2.2 Topologie *fat-tree* . . . 7
2.3 Écriture d'un nouvel algorithme d'ordonnancement . . . 7

3 Contexte technologique **9**
3.1 Fonctionnement de MPTCP . . . 9
3.2 Utilisation réelle de MPTCP . . . 11
3.3 MPTCP et sécurité . . . 12

4 Analyse **13**
4.1 Topologies virtualisées . . . 13
4.1.1 Multi-chemins simple . . . 13
4.1.2 FatTree . . . 13
4.1.3 MPTCP vs TCP . . . 13
4.2 Performances de MPTCP . . . 14
4.3 Conception algorithme sécurisé . . . 14

5 Conception **16**

6 Compte rendu du projet **17**
6.1 Outil de coordination : git . . . 17
6.2 Compilation MPTC . . . 17
6.3 Topologie . . . 17
6.3.1 Topologie simple . . . 17
6.3.2 FatTree . . . 18
6.4 Performance de MPTCP sur mininet . . . 20
6.4.1 Un exemple de démonstration . . . 20
6.4.2 Variation du débit maximal par lien . . . 21
6.4.3 Variation du délai par lien . . . 23
6.4.4 Choix du sous-flot en fonction du délai . . . 25
6.5 Etude de l'algorihme de MPTCP . . . 30
6.5.1 Explication de l'ordonnanceur . . . 30
6.6 Choix de l'ordonnanceur . . . 31
6.7 Implémentation de l'ordonnanceur . . . 32

7 Conclusions **34**

8 Perspectives **34**

9 Remerciements **34**

10 Annexe1 **35**
10.1 Utilisation des scripts pythons . . . 35
10.1.1 Activation MPTCP . . . 35
10.1.2 Choix de l'algorithme congestion . . . 35
10.1.3 Taille de la fenêtre . . . 36
10.2 Lancement scripts python . . . 36
10.3 Résumé des arguments pour le parseur . . . 37
10.4 arguments mininet . . . 38
10.4.1 ssh . . . 38
10.5 scripts shell . . . 39
10.6 Scripts R . . . 40
10.7 Bugs . . . 40
10.7.1 Topologie . . . 40
10.7.2 mininet . . . 40

1 Cahier des charges

1.1 Objectifs

Les objectifs du projet sont de :

- mesurer les performances de MPTCP sur différentes topologies de réseaux virtuels.
- modifier l'ordonnanceur de MPTCP pour privilégier une répartition équilibrée sur les différents sous-flots.

1.2 Contexte

MPTCP est une extension de TCP qui permet pour une connexion TCP donnée d'utiliser plusieurs chemins pour l'échange de données. La multiplicité des sous-flots a pour but d'améliorer le débit et d'augmenter la résilience de la connexion [1–3].

Les performances de MPTCP ne doivent pas être inférieures à celles de TCP et son l'utilisation ne doit pas diminuer le débit des autres utilisateurs sur le méme réseau. Les performances de MPTCP dépendent en partie de l'algorithme utilisé pour la répartition des données entre les différents sous-flots ouverts [4]. Pour caractériser les performances de l'ordonnanceur de MPTCP, nous allons le tester dans différents réseaux virtualisés en utilisant dans un premier temps l'algorithme implémenté dans le kernel MPTCP de linux [1].

L'emploi de MPTCP améliorerait la sécurité si les données transitaient de manière équilibrée entre les différents sous-flots, ce qui complexifient les attaques. Le débit global de la connexion serait affecté car les chemins les plus lents vont ralentir le débit des chemins les plus rapides, ce qui, en contre partie, peut s'avérer moins performant qu'une simple connexion TCP. Nous allons modifier l'ordonnanceur afin de garantir la répartition équitable des charges puis analyser l'influence de cette modification sur les performances de MPTCP dans les topologies réseaux utilisées précédemment.

1.3 Méthodes

La réalisation du projet peut être subdivisée en trois parties :

- la simulation de réseaux à topologies différentes,
- l'analyse des performances de MPTCP,
- l'adaptation de l'ordonnanceur pour l'aspect sécurité.

Nous utiliserons Mininet afin de simuler les topologies réseaux où nous pourrons mesurer les performances de MPTCP à l'aide de l'API Python fournie par Mininet. Pour caractériser l'influence de l'ordonnanceur sur les performances, nous utiliserons des réseaux simples où les différents sous-flots sont asymétriques et diffèrent par une propriété à la fois : latence, débit, pertes... Nous testerons différents algorithmes de répartition de charge entre sous-flots : l'algorithme implémenté par défaut (LIA), celui qui satisfait l'op-

1. mptcp.org

timum de pareto par rapport aux objectifs de MPTCP [1,4], ou encore un algorithme de répartition équilibrée de la charge réseau entre les différents sous-flots.

2 Plan de développement

La première partie est de consuitre les topologies de différents réseaux virtuels et de tester les performances de MTPCP en faisant varier les paramètres des sous-flots. La seconde partie est de construire un algorithme d'ordonnancement répondant à un critère de sécurité simple : rendre le *sniffing* des paquets plus difficile à réaliser.

Les étapes du développement suivront les points suivants :
- Préparation d'une machine mininet avec le noyau MPTCP compilé pour l'ensemble de l'équipe :
- Lecture, compréhension et commentaires du code de MPTCP ;
- Préparation de plusieurs topologies : une topologie simple à plusieurs nœuds et un *fat tree* pour simuler un *data center* ;
- Préparation d'une bibliothèque de tests et de mesures via l'API python ;
- Écriture d'un algorithme d'ordonnancement dans le noyau ;
- Mesures de performance.

Le développement sera divisé en trois sous-projets :

Scripts de tests mininet Le but de ce sous-projet est d'établir une base pour tester les performances de MPTCP en fonction des paramètres du réseau virtuel. Cette base sera réutilisée pour les topologies plus complexe (sous-projet *fat-tree*) ou pour la mesure de performances du nouvel algorithme d'ordonnancement. (sous-projet algorithme d'ordonnancement).

Responsable du sous-projet : M. Ly.
- Création d'une machine virtuelle contenant le noyau MPTCP ;
- écriture des scripts pour les simulations et l'analyse ;
- tests sur une topologie simple.

fat tree Ce sous-projet vise à mesurer les performances sur une topologie complexe où MPTCP a une utilité substantielle.

Responsable du sous-projet : M. Ravier.
- Création d'une topologie *fat tree* ;
- tests.

Écriture d'un algorithme d'ordonnancement Ce sous-projet vise à c

Responsables du sous-projet : M. Lam et M. Dubois.

- Création d'une topologie *fat tree* ;
- tests.

2.1 Scripts de tests mininet

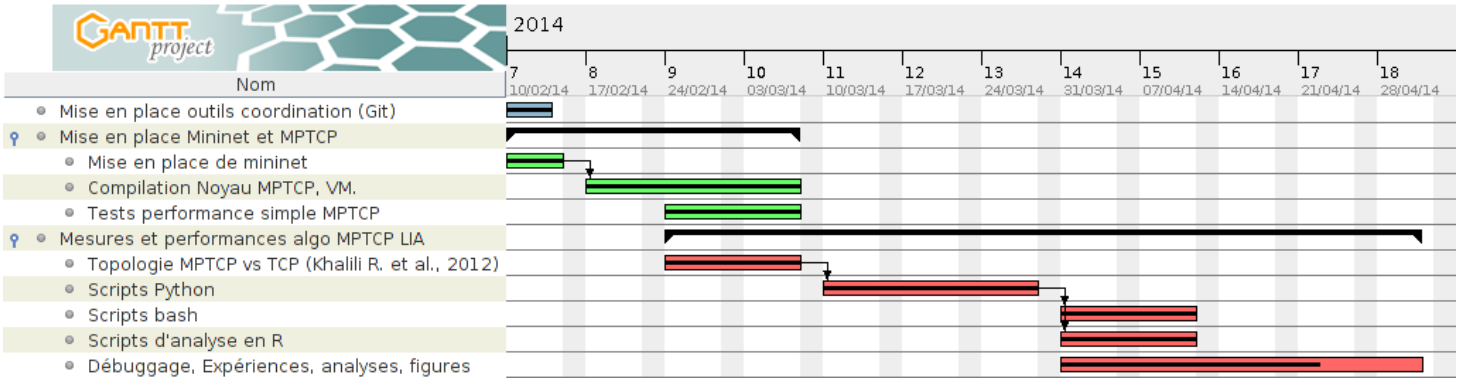


Figure 1 – **Diagramme de Gantt de Romain Ly.**

- Mise en place du Git
- Compilation noyau MPTCP dans la VM mininet et tests simples de routines
- Topologie modifiée de Khalili et al.
- Scripts Python
  - implémentation d'un parseur d'arguments
  - intégration de ping, iperf, iperf3, bwm-ng, sshd, tcpdump permettant le monitoring et l'étude du réseau
- Installation de TCP-reduce dans la VM pour vérifier les options de la connexion TCP.
- Scripts Bash pour générer de multiples simulations par l'intermédiaire de scripts Python
- Scripts R pour analyse des résultats et figures
- Débogages des scripts
- Expérimentations
  - variation du nombre de sous flots, débits, latences, taille de fenêtre, etc.

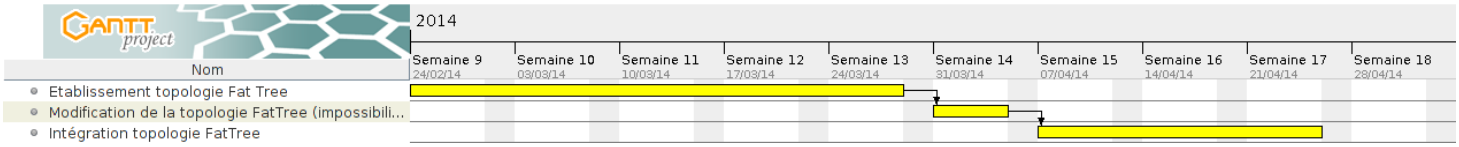


FIGURE 2 – **Diagramme de Gantt de Simon Ravier.**

2.2 Topologie *fat-tree*

03/03 au 30/03	établissement de la topologie FatTree
31/03 au 06/04	modification de la topologie FatTree (impossibilité technique de réaliser le premier modèle)
07/04 au 27/04	Intégration de la topologie FatTree dans l'environnement de tests (configuration des hôtes, établissement des tables de routage) et réalisation des tests

2.3 Écriture d'un nouvel algorithme d'ordonnancement

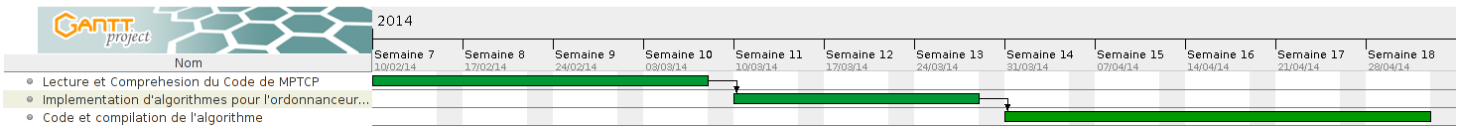


FIGURE 3 – **Diagramme de Gantt Kevin Lam et Quentin Dubois**.

Analyse de la structure de l'implémentation :
Déterminer globalement les fichiers à lire ;
Définir les headers et structures liés à l'utilisation de mptcp.

Lecture et Compréhension du code :
Lecture de tous les fichiers contenus dans $(SRC_NOYAU)/net/mptcp ;
Lecture de certains fichiers contenus dans $(SRC_NOYAU)/net/sched ;
Lecture de tous les types/structures utilisés ;
Relecture du code après avoir compris tous le types/structures.

Définition des parties modifiables de l'ordonnanceur :
Vérifier les correspondances entre mptcp_output.c et mptcp_input.c ;

Voir si l'utilisation du contrôle de congestion (mptcp_olia.c et mptcp_coupled.c) a des effets de bord sur les sockets ;
Voir les différences entre les différents modes du path_manager.

Implémenter / Tests d'algorithmes pour l'ordonnanceur
Avec l'aide de Matthieu Coudron, modification dans $(SRC_NOYAU)/net/mptcp/mptcp_output.c ;
Tests des implémentations avec le travail de Romain LY et comparer les résultats obtenus.

3 Contexte technologique

L'élaboration de MPTCP a été motivée par l'observation de l'existence dans les réseaux de plusieurs chemins entre deux machines A et B. L'utilisation de ces différents chemins entre les deux hôtes pourrait être un atout non négligeable pour augmenter le débit de la connexion et/ou la résilience de la connexion si l'un des chemins venaient à ne plus pouvoir acheminer les paquets (congestion, panne de routeur, etc). De plus le multi-chemin permet d'équilibrer la répartition des charges sur les sous-flots utilisés. TCP n'a pas été conçu pour exploiter plusieurs chemins d'où la nécessité de concevoir des protocoles multi-chemins comme MPTCP permettant d'utiliser les chemins disponibles pour transmettre les paquets d'une connexion entre A et B via les sous-flots connectés.

Il existe déjà plusieurs protocoles proposant d'utiliser plusieurs chemins. Nous en citerons que deux : SCTP et ECMP. SCTP (*Stream Control Transmission Protocol*) allie l'avantage de TCP et UDP (utilisation de datagrammes) et permet de multiplexer les flux sur plusieurs interfaces [5]. ECMP (*Equal Cost MultiPath*) est un sous-protocole dans le cadre de divers protocoles de routages (comme OSPF, TRILL, ...) et qui est utilisé dans les *data-centers* : lors d'une connexion entre deux hôtes, le routeur peut transférer les paquets sur plusieurs meilleurs chemins à coûts « égaux » [6].
Les inconvénients de SCTP est la nécessité que tous les hôtes terminaux puissent comprendre le protocole ; il est donc nécessaire de modifier la couche application pour pouvoir l'utiliser. De plus, les *middle boxes* (pare-feu, NAT, ...) ne reconnaissent pas le protocole et rejettent donc tous les paquets SCTP. ECMP utilise les routeurs pour connaître les chemins et l'augmentation de performance lié à son utilisation n'est pas significative.

L'avantage principal de MPTCP est d'être rétrocompatible par rapport à TCP. Si un hôte n'est pas compatible avec MPTCP, la connexion utilisera TCP résolvant les problèmes des *middle boxes*. Le scond avantage est qu'il est totalement transparent pour les routeurs, c'est une connexion *end to end*.

3.1 Fonctionnement de MPTCP

MPTCP utilise dans un premier temps une connexion TCP pour créer des sous-flots similaires à TCP avec des chemins différents. La couche TCP est alors remplacée par la couche MPTCP qui est divisée en deux parties (Fig. 4) : la couche supérieure correspond aux fonctions nécessaires à MPTCP de fonctionner (découverte et gestion des chemins, ordonnancement des paquets, contrôle de congestion) et la couche inférieure correspond aux sous-flots établis.

MPTCP permet l'augmentation du débit de manière à ce qu'il soit supérieur à une connexion TCP unique sur le meilleur des chemins disponibles. Il permet aussi la résilience de la connexion en cas de panne ou de congestion, dans ce cas le trafic est alors réparti et/ou réexpédié sur les autres sous-flots sans la nécessité de rétablir une connexion TCP entre les deux terminaux. Cette approche permet de répartir les charges sur les ressources disponibles.

MPTCP implémente plusieurs fonctions pour contrôler les sous-flots de la connexion

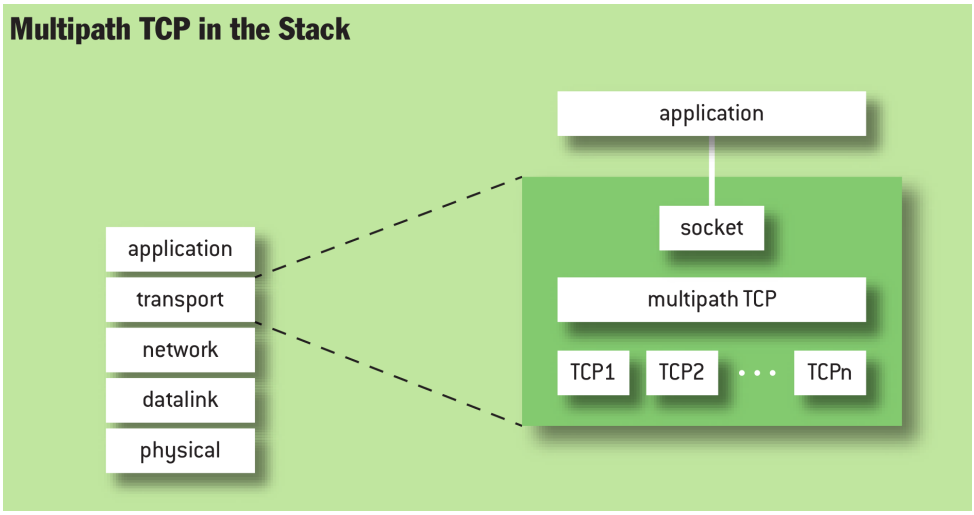


Figure 4 – **Schéma de la pile MPTCP** [7].

[1] :

- le gestionnaire de chemin ou *path manager*
- l'ordonnanceur de paquets ou *packet scheduling* puis
- le contrôleur de congestion ou *congestion controller*.

Le sous-flot prend en charge les segments de l'ordonnaceur pour l'envoyer sur un chemin disponible. Le sous-flot agit comme une connexion TCP classique et dispose de cette manière des fonctions de ce protocole de transport assurant d'envoyer des paquets de manière fiable et séquencée. À la réception, le sous-flot envoie les segments à la couche ordonnanceur de paquet pour le réassemblage.

Le gestionnaire des chemins est le mécanisme permettant de détecter et d'utiliser les chemins disponibles par l'intermédiaire de multiples adresses IPs dans les hôtes. Il signale l'existence d'adresses alternatives et permet d'intégrer de nouveaux sous-flots à une connexion MPTCP existante ou d'en enlever.

L'ordonnanceur des paquets découpe le flux de données provenant de la couche application en segments prêts à être envoyés par l'un des sous-flots. Il séquence les segments et permet de réassocier les segments pour réordonner les données côté destinataire. L'ordonnanceur dépend des informations des chemins disponibles provenant du gestionnaire de chemins [8].

Enfin, le contrôle de congestion est un outil essentiel qui permet d'adapter le débit de chaque sous-flot et de définir si un chemin est trop lent par rapport au meilleur sous-flot. Il permet aussi de renvoyer l'information au gestionnaire s'il y a une panne.

Le contrôle de congestion nécessite un algorithme performant pour que l'utilisation de MPTCP à la place de TCP puisse effectivement augmenter le débit de l'utilisateur sans influencer le débit des autres utilisateurs sur les mêmes chemins, c'est à dire qu'il doit garantir l'optimalité de pareto (c'est à dire que l'allocation des ressources est réalisée de

manière ce qu'il soit impossible d'augmenter le débit d'un utilisateur sans diminuer le débit d'un tiers ou sans augmenter le coût de la congestion. Il garantit aussi la répartition équitable de la capacité du lien entre les utilisateurs [4]). L'algorithme de MPTCP est donc un point central dans les performances de MPTCP sur le réseau.

Lors des choix des sous-flots, l'algorithme doit effectuer un compromis entre équilibre des charges dans les différents sous-flots et réactivité (*responsiveness*) en cas de modification de la latence des sous-flots ou de découverte de nouveaux chemins. Une priorité vers l'équilibre des charges entraîne l'envoi des données sur les meilleurs routes (selon la métrique utilisée, par défaut la latence du chemin) mais cela peut déclencher un changement constant de route produisant un effet de battement (*flappiness*) : si plusieurs chemins possèdent le même coût, l'algorithme aura tendance à changer plus souvent de chemins. Si la priorité utilisée est la réactivité (par augmentation de la taille de la fenêtre d'un des sous-flot), l'utilisation de toutes les ressources disponibles peut ne pas être optimale car on aura tendance à utiliser qu'un seul sous-flot. Les paramètres de l'algorithme doivent être déterminés efficacement pour répartir les charges sur les sous-flots et ne pas être agressif (augmentation trop rapide de la taille de la fenêtre sur un des sous-flots) pour garantir l'optimalité de Pareto [4].

Dans l'algorithme par défaut, le critère privilégié par l'algorithme est le RTT. Il serait intéressant de modifier les caractéristiques du réseau pour mesurer les performances de MPTCP sur le choix des chemins utilisés ou en cas de modification de chemins sur des critères de latence, pertes, débit ...

3.2 Utilisation réelle de MPTCP

Dans la pratique, l'utilisation de MPTCP est difficile. L'utilisation de plusieurs sous-flots ne garantie pas l'augmentation de débit. Pour celà, il est nécessaire que les sous-flots empruntent des chemins physiques différents et aujourd'hui il n'est pas possible pour un utilisateur de contrôler le routage de ses paquets de bout en bout. Une méthode pour contourner le problème serait d'utiliser la conjonction de MPTCP et de LISP (*Locator/Identifier Separation Protocol*) qui permet de découvrir la diversité de chemins existant entre routeurs de bordures (A-MPTCP) [3].

Cependant il existe des cas où MPTCP est utilisable à son plein potentiel et suscite l'intérêt : dans les datacenters et en environnement mobile. Par l'intermédiaire d'une stratégie de routage par SDN (*Software Defined Network*) par exemple OpenFlow, le contrôleur peut établir des chemins différents entre deux hôtes sur tout son réseau. Le transfert de données au sein d'un datacenter nécessite des débits très importants. L'utilisation de MPTCP pourrait répartir les charges entre les différents noeuds. Des expériences sur différentes topologies de datacenter à haute densité ont permis de montrer que MPTCP égale, voir surpasse même la performance d'un ordonnanceur centralisé et est de surcroit plus robuste [9]. En mobile, le terminal pourra utiliser le réseau 3G/4G et le réseau wi-fi environnant. MPTCP permettra de décharger le réseau téléphonique de l'opérateur tout en augmentant le débit et la résilience de la connexion.

3.3 MPTCP et sécurité

À l'heure d'Eric Snowden, l'utilisation de plusieurs sous-flots pourrait être un avantage non négligeable en terme de sécurité. Pour pouvoir épier une connexion entre A et B, il faudrait à l'attaquant de pouvoir *sniffer* les paquets qui sont émis sur les sous-flots utilisés, c'est-à-dire sur autant de chemins physiques différents. L'intérêt du multi-chemin prend alors tout son sens. Cependant ce n'est pas le seul avantage, on peut réfléchir à plusieurs moyens d'augmenter la sécurité par l'utilisation du multi-chemin conjoitement avec une modification des protocoles de sécurité. Voici quelques idées personnelles d'une implémentation d'une combinaison de MPTCP et de sécurité.

- l'utilisation d'une méthode de chiffrement par bloc de type CBC (*Cipher Block Chaining*) compliquera la tâche de l'attaquant car il sera nécessaire d'obtenir le bloc n-1 pour déchiffrer le bloc n. Par exemple, AES-CBC est utilisé couramment dans des communications utilisant SSL/TLS. L'attaquant devra disposer de tous les paquets sans exception pour pouvoir déchiffrer la totalité du message en supposant qu'il possède la clé secrète.
- De plus, si les protocoles de sécurité sont conscients de l'utilisation de MPTCP, il pourrait y avoir une entente *cross-layer*. Par exemple, en distribuant les informations des MAC (*Message Authentication Code*) de chaque paquet entre les différents sous-flots de manière à éviter les *man in the middle* : sous flot 1 = message 1 + HMAC (message 2) ; sous flot 2 = message 2 + HMAC (message 1).
- Un autre exemple serait de négocier les clés pour le chiffrement de la communication d'un sous-flot (par exemple en utilisant IPSec) dans le sous-flot adjacent.

Dans tous les cas, il est donc nécessaire que MPTCP dans une optique sécurité utilise au mininum deux sous-flots et donc de modifier l'algorithme de congestion de MPTCP. Dans une première approche simpliste, il serait intéressant de forcer l'algorithme de MPTCP à répartir les paquets équitablement sur plusieurs sous-flots, quitte à diminuer les performances de MPTCP, c'est l'idée que nous avons décidé d'utiliser pour le nouvel algorithme d'ordonnancement.

4 Analyse

4.1 Topologies virtualisées

Nous allons simuler des topologies openFlow en utilisant mininet. Les switchs seront virtualisés par openvSwitch (OVS) qui est installé par défaut dans l'image mininet [1]. Pour utiliser le multi chemin, le noyau de MPTCP sera compilé dans la machine virtuelle et chaque hôte sera configuré de manière adéquate pour pouvoir utiliser MPTCP.

Nous créerons et testerons les topologies virtuelles grâce à l'API python.

4.1.1 Multi-chemins simple

La topologie simple est composée de deux hôtes et de N switchs. Les N switchs composeront les N chemins disponibles. Cette topologie simple servira principalement au test de fonctionnement de MPTCP.

4.1.2 FatTree

Afin de tester MPTCP de manière réaliste, nous avons simulé une topologie FatTree, souvent utilisée dans les datacenters qui sont les premiers demandeurs des performances offertes par MPTCP. Cette topologie repose sur le principe d'établir plusieurs liens physiques entre deux équipements réseau, en l'occurrence des switchs. Tous les switchs du réseau ont le même nombre de ports ; ils sont organisés par couches : une couche « coeur », une couche « frontière » et une couche « hôtes ». Les couches hôtes et coeurs sont directement connectées à la couche frontière, mais pas entre elles. Chaque switch de la couche coeur est connecté à chaque switch de la couche frontière par de multiples liens. Le nombre de ports disponibles sur les switchs coeurs est équitablement réparti entre chaque switch frontière ; ainsi, avec deux switchs coeurs et quatre switchs frontières à 36 ports, on disposera de 9 liens entre chaque paire de switchs de couches différentes. Le reste des ports disponibles sur les switchs frontières sont utilisés pour y connecter les hôtes, à raison d'un lien par hôte. Notons que deux équipements d'une même couche ne sont jamais interconnectés.

4.1.3 MPTCP vs TCP

Pour déterminer les critères à respecter de l'ordonnanceur et conserver les principes de MPTCP (équitabilité avec les utilisateurs TCP et performances supérieures à TCP), nous allons reproduire le *testbed* utilisé dans l'article de Khalili (Fig. 5) [4].

Si nous pouvons reproduire les résultats obtenus par Khalili avec notre configuration, nous reproduirons le cas avec N1 utilisateurs MPTCP et N2 utilisateurs TCP (voir Fig. 6).

1. http://mininet.org/walkthrough/#other-switch-types

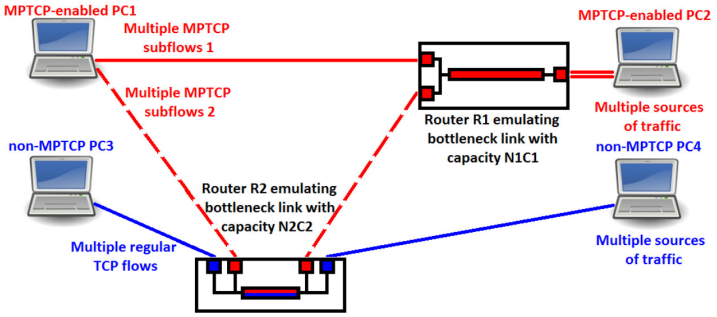


Figure 5 – **Testbed MPTCP vs TCP [4]**.

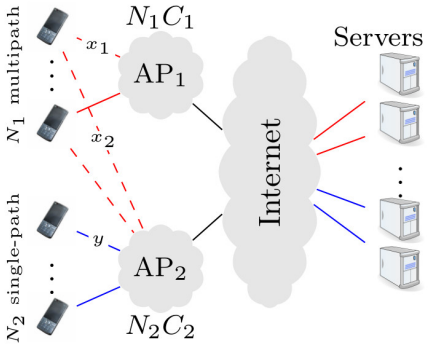


Figure 6 – **Testbed MPTCP vs TCP [4]**. Les N1 utilisateurs MPTCP (rouge) utilisent deux points d'accès pour se connecter à un serveur distant dont un qui est partagé avec les N2 utilisateurs TCP (bleu).

4.2 Performances de MPTCP

Pour mesurer les performances de MPTCP, nous allons faire varier les propriétés de chaque sous-flot emprunté en modifiant les chemins de manière asymétrique. Le but est de créer des conditions de stress qu'on pourra tester à la volée avec les différents algorithmes gérant MPTCP (celui par défaut, l'OLIA et le notre si celui-ci est opérationnel) et sur les différentes topologies virtuelles construites.

Les contraintes appliquées auront comme critères la latence (critère actuellement privilégié par l'ordonnanceur pour les choix de sous-flots), la capacité, le taux d'erreur, la gigue, etc. Nous testerons quelle est l'influence de ces paramètres sur le choix des sous-flots par l'ordonnanceur.

4.3 Conception algorithme sécurisé

Le but est de rendre une connexion plus sécurisée par la complexité de l'analyse des paquets de données échangés entre deux utilisateurs. Nous chercherons à faire une méthode simple et non performante pour effectuer des tests et savoir comment MPTCP réagit au

nouvel algorithme de répartition des paquets dans les sous-flots TCP. Cette méthode consiste à prendre le nombre de sous-flots total et de répartir les segments équitablement entre les différents sous-flots. Le débit de chaque sous-flot correspondra au débit le plus faible des sous-flots. Cela reste une solution de l'objectif noté dans le cahier des charges.

Néanmoins il sera nécessaire d'avoir un algorithme plus intelligent. En effet, il est nécessaire d'avoir un meilleur algorithme que celui expliqué ci-dessus car la performance pourrait être grandement affectée. Le débit pourrait être bien plus faible qu'une connexion TCP classique sur le meilleur des chemins si un des chemins a un débit beaucoup plus faible ou s'il est congestionné. Or même si nous voulons accroître la sécurité il est préférable d'avoir au moins le débit d'une connexion TCP simple. La difficulté dans cette partie est de pouvoir adapter l'ordonnanceur selon le nombre de sous-flots disponibles. Une idée simple serait de répartir les charges de manière à que l'ordonnanceur de paquets n'envoie plus de $50 + \varepsilon$ % des segments à un unique sous-flot. Ce nombre est arbitraire, il pourrait être variable selon d'autres paramètres comme le nombre de sous-flots disponibles et il faudrait définir un algorithme global pour le déterminer.

5 Conception

5.1 Outil de coordination : git

L'Ã©tat des scripts utilisÃ©s par l'Ã©quipe est mise Ã jour par un systÃ¨me de version utilisant git https://github.com/Romain-Ly/PRES. Le noyau contenant les modifications de MPTCP est disponible ici : https://github.com/Finaler/mptcp/tree/mptcp_pres_test1

5.2 Mininet

Nous avons utilisÃ© trois langages pour les scripts.

python Dans un premier temps, nous avons utilisÃ© python pour pouvoir intÃ©grer les fonctions de l'API python et remplir les fonctions suivantes :

- crÃ©er des topologies
- crÃ©er des expÃ©riences variÃ©es
- intÃ©grer des outils pour la mesure des performances
- intÃ©grer un parseur d'argument pour automatiser les procÃ©dures

L'axe d'Ã©criture des scripts est de pouvoir intÃ©grer des topologies, des expÃ©riences Ã la volÃ©e et de permettre d'Ã©crire des fichiers de sorties diffÃ©renciÃ©s afin de rendre l'analyse plus facile. Une expÃ©rience peut se dÃ©finir comme l'ensemble des contraintes apportÃ©s Ã la topologie (dÃ©lai d'un ou de plusieurs liens, tests utilisÃ©es, dÃ©roulement des mesures).

Une description plus dÃ©taillÃ©e est effectuÃ©e dans le compte-rendu (voir 6 page 17) et dans l'annexe (voir 10 page 35).

bash Pour multiplier les expÃ©riences, nous utilisons des scripts *bash* exÃ©cutant les scripts pythons en sÃ©rie.

R Pour analyser les fichiers de sortie (iperf, bwm-ng, ping, ...) et pour les figures nous avons utilisÃ© R.

5.3 Performances de MPTCP

5.4 Conception algorithme sÃ©curisÃ©

AprÃ¨s avoir Ã©tudiÃ© le code de MPTCP, nous nous sommes demandÃ©s comment nous pouvions avoir un rendement maximal afin d'implÃ©menter l'ordonnanceur. Nous avons donc discutÃ© de nos mÃ©thodes d'implÃ©mentation pour essayer de s'accorder mais comme nos idÃ©es diffÃ©raient sur la faÃ§on de procÃ©der pour la rÃ©alisation de cet ordonnanceur, nous avons dÃ©cidÃ©, dans un premier temps, de coder un ordonnanceur chacun de notre cÃ´tÃ©. Une fois que nos ordonnanceurs ont

Ã©tÃ© codÃ©s, nous avons confrontÃ©s nos versions et choisi de n'en garder qu'une seule car elle semblait mieux structurÃ©e ce qui Ã©tait un atout si nous devions apporter des modifications Ã cet algorithme.

AprÃ¨s avoir fini notre ordonnanceur, nous avons constatÃ© qu'il y avait des erreurs dans le code que nous avons corrigÃ©s et ensuite synchronisÃ© grÃce au git. Une fois que notre code compilait, nous avons du tester si il effectuait se que nous voulions. Pour cela, nous nous sommes encore partagÃ© le travail en testant le code chacun de notre cÃ´tÃ© et en utilisant diffÃ©rents moyens afin de comprendre ce qui ne marchait pas dans notre algorithme.
Afin de ne pas perdre de temps et de rester le plus productif possible, nous partagions tout ce que nous avions rencontrÃ©s avec les autres membres du groupe afin de savoir si quelqu'un avait dÃ©jÃ eu ce problÃ¨me ou pour nous donner une idÃ©e de rÃ©solution.

6 Compte rendu du projet

6.1 Compilation du noyau MPTCP (machine test)

Pour la réalisation du projet, nous utilisons une machine virtuelle ubuntu 13.04 (32 bits) où mininet est installé et est prêt pour utilisation : https://bitbucket.org/mininet/mininet-vm-images/downloads.

Nous avons compilé le noyau linux contenant MPTCP (v0.88) dans une VM de mininet (v2.10). Les paquets debian pour l'installation du noyau MTPCP sur une VM de mininet est disponible : (https://www.dropbox.com/sh/y4ykck8rg6908ps/7V3lsV6Ggg).

Cependant, l'architecture des machines pouvant être différente, la VM n'est pas fonctionnelle sur certaines des machines des membres du projet. Par conséquent, il est préférable de compiler soit même MPTCP dans chaque machine. Le fichier ".config" utilisé pour la compilation est disponible sur le premier git. La seconde option est d'utiliser "apt-repository" selon les indications disponibles à cette adresse http://multipath-tcp.org/pmwiki.php/Users/AptRepository.

6.2 Topologie

6.2.1 Topologie simple

La première topologie est inspirée du *testbed* de l'article de R. Khalili [4], voir Fig. 7. Nous avons ajouté, à cette topologie, des routeurs privés entre le client et le serveur *MPTCP* pour disposer d'un nombre de sous-flots supérieur à deux.

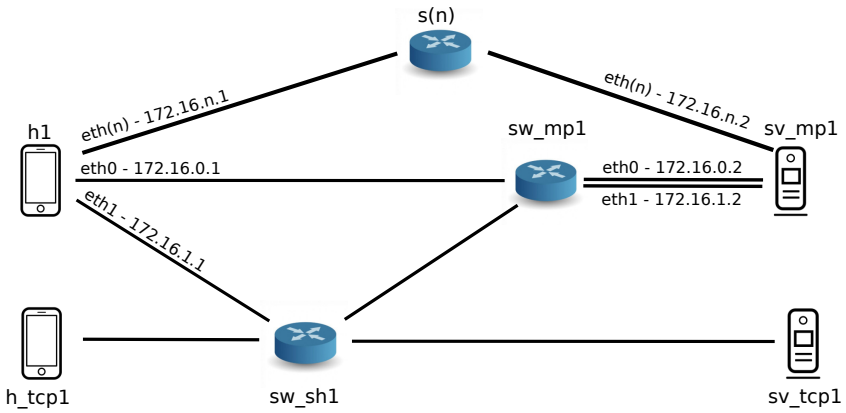


Figure 7 – **Reproduction de la topologie de l'article de R. Khalili**. Le(s) *switch(s)* "S(n)" ne sont présent(s) que si le nombre de sous-flot est supérieure à deux. Pour n sous-flots, il y aura $n-2$ *switchs* et $n-2$ liens supplémentaires. L'hyperviseur est connecté à tous les switchs. Pour se connecter via ssh aux hôtes, un *switch* « root » est créé et est connecté au *switch* sw_mp1 (non représenté ici) voir utilisation CF linktobeadded.

6.2.2 FatTree

Nous avions le choix entre plusieurs topologies pour réaliser des tests plus « réalistes ». Bcube, VL2, FatTree sont autant de topologies utilisées dans les *data center* aujourd'hui. Pour des raisons de facilités techniques, nous avons choisi d'implémenter une topologie FatTree. Suite à notre recherche de documentation, nous nous sommes retrouvés confrontés à un choix : plusieurs définitions du FatTree sont ressorties, et certains détails constituaient de très nettes différences au niveau de l'implémentation selon le modèle choisi. La simplicité d'implémentation de cette topologie repose sur le fait qu'il s'agit d'un arbre dont la connectivité entre les nœuds augmente lorsqu'on se rapproche de sa racine. Pour commencer, nous nous sommes proposé de choisir l'approche la plus générale possible en considérant un FatTree à 3 niveaux (Core, Edge, Hosts) :

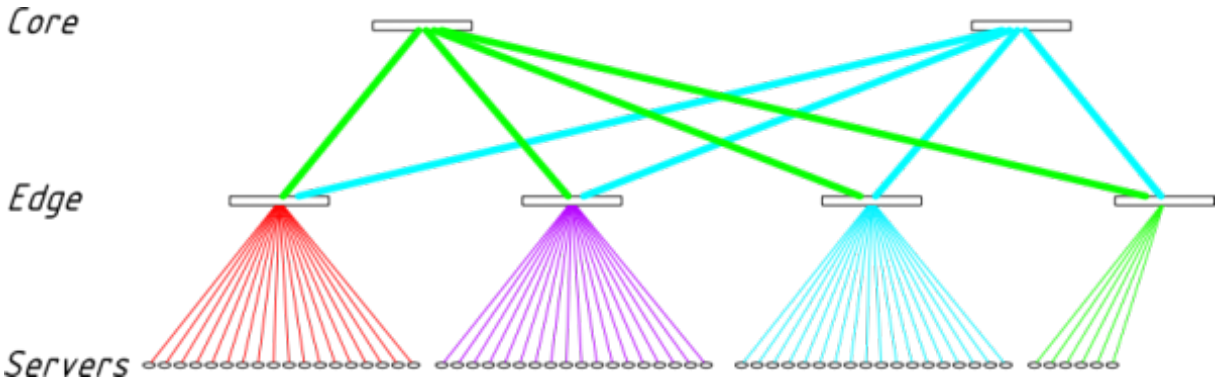


Figure 8 – **Modélisation d'une topologie *Fat-tree***

Cette modélisation considère que les switches sont tous identiques, et possèdent tous 36 ports. Chacun des 2 switches du niveau racine (*Core switches*) dispose du même nombre de liens vers chacun des 4 switches du niveau intermédiaire, à savoir 9 liens vers chacun d'eux (*Edge switches*). Ainsi, la moitié des ports de chaque *Edge* switch, à savoir 18, sont dédiés au niveau supérieur. Cela laisse donc l'autre moitié pour y connecter autant d'hôtes, à raison d'un lien chacun. Le réseau peut donc atteindre un maximum de 72 hôtes. Cette représentation a été choisie en raison de sa grande flexibilité. On peut aisément manipuler l'envergure du réseau de test (nombre d'hôtes, diversité des chemins d'un hôte à un autre) en fonction du nombre de *Core switches* ou de *Edge switches* ainsi que du nombre de ports sur chaque switch. Cependant, nous nous sommes heurtés à une difficulté technique lors de l'implémentation de cette version du *FatTree*. En effet, mininet ne supporte pas les liens multiples entre switches, ce qui a résulté en une diversité de chemins entre hôtes insuffisante pour donner de l'intérêt à une simulation de MPTCP. Nous avons donc du revoir nos plans et nous pencher sur un nouveau modèle de *FatTree*.

Dans la nouvelle version de notre topologie, il n'y a plus lieu de parler de multiples liens entre switches ; ils sont tous interconnectés par des liens uniques. Le problème de la

diversité des chemins est résolu par la transformation du *FatTree* à 3 niveaux en un *FatTree* à 4 niveaux. Nous avons effectué la séparation du niveau intermédiaire (*Edge*) en 2 sous-niveaux. Les switches connectés au niveau 1 (*Core*) seront appelés *Aggregation switches*, et ceux connectés aux hôtes resteront les *Edge switches*. La topologie ainsi obtenue est schématisée ci-dessous.

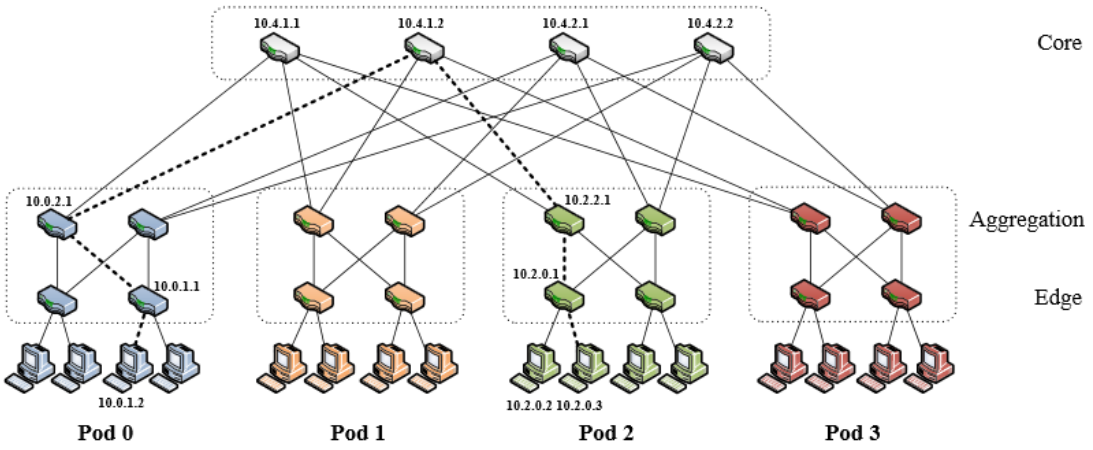


Figure 9 – **Modélisation d'une topologie *Fat-tree***

Nous avons désormais 4 Core switches, 8 switches de niveau 2 (*Aggregation*) et autant au niveau 3 (*Edge*), ainsi que 2 hôtes pour chaque *Edge switch* pour un total de 16 hôtes. L'interconnexion entre les sous-niveaux 2 et 3 est quelque peu particulière : sont regroupés par clusters de 2 les *Aggregation switches*, puis associés à un cluster de 2 *Edge switches*. On obtient ainsi 4 clusters de 4 switches répartis sur les niveaux 2 et 3. Dans chacun de ces clusters, les 2 *Edge switches* sont connectés aux 2 *Aggregation switches* (il n'y a bien sûr pas d'interconnexion entre switches de même niveau, il s'agit d'une structure arborescente), résultant en une duplication du nombre de chemins possibles. De plus, les *Aggregation switches* sont chacun connectés à 2 des 4 *Core switches*, dupliquant à nouveau le nombre de chemins possibles. En résulte que chaque cluster peut s'assimiler à un *Edge switch* du modèle précédant, offrant une connectivité directe vers les hôtes autant que vers chacun des *Core switches*.

6.3 Performance de MPTCP sur mininet

Après la compilation du noyau, pour vérifier le fonctionnement de MPTCP nous avons mesuré le débit moyen en utilisant *iperf* sur la topologie A.

Les paramètres [1] utilisés sont les suivants :

1. paramètres par défaut dans le noyau Linux

Paramètre	Valeur
MSS	1460 octets [2]
window size	85,3 Koctets
délai par lien	10 ms
Algorithme de congestion	LIA [8]

6.3.1 Un exemple de démonstration

Nous allons prendre, dans cet exemple, une connexion avec deux sous-flots avec une capacité individuelle de 100 Mbit/s. Voici la commande pour générer cet exemple :

```
sudo python ./pyMPTCP -O exp001_TC --bw 100 -t 30 -n 2 --mptcp --bwm_ng
```

Pour les détails de l'activation de MPTCP dans le noyau et l'utilisation des arguments des scripts python, une notice est donnée en Annexe 1 : voir sections 10.1 page 35 et 10.3 page 37.

Le RTT entre h1 et sv_mp1 est de 44 ± 11 ms (mesuré avec la commande *ping*), ce qui correspond au RTT attendu pour traverser deux liens aller-retour. La moyenne ici tient compte du RTT du premier paquet qui est envoyé vers le contrôleur pour que celui-ci établisse le chemin vers le serveur.

L'argument "--bwm_ng" permet de lancer Bandwidth Monitor NG [3] (bwm-ng). Cette application mesure plusieurs paramètres : le nombre d'octets, de paquets, le débit entrant ou sortant passant par chacune des interfaces de l'hôte sondé. La figure 10 représente le débi entrant enregistré au serveur pour ses deux interfaces. Nous observons pour deux sous-flots aux capacités identiques et au coût identique, que le débit mesuré est quasi similaire (une différence de quelques paquets est observée).

Pour mesurer le débit total généré, nous effectuons une moyenne des débits totaux mesurés toutes les secondes par *iperf* entre 5 secondes après le début de la connexion, et 1 seconde avant la fin de la connexion. Nous observons pour cet exemple, côté serveur, un débit maximal de 168 Mbit/s. Ce débit est cohérent par rapport aux mesures de débit *via* bwm-ng.

6.3.2 Variation du débit maximal par lien

Pour connaître les limites de nos simulations, nous avons fait varier la capacité de chaque sous-flot, ainsi que le nombre de sous-flots dans la topologie A.

Le débit totale mesuré au serveur croît linéairement avec l'augmentation du nombre de sous-flots pour des capacités par lien de 10 à environ 100 Mbit/s. Cette phase est en adéquation avec le but de MPTCP : augmentation des performances [1]. Cependant, pour les liens aux capacités supérieures à 100 Mbit/s, les performances décroissent rapidement

3. http://www.gropp.org/?id=projects&sub=bwm-ng

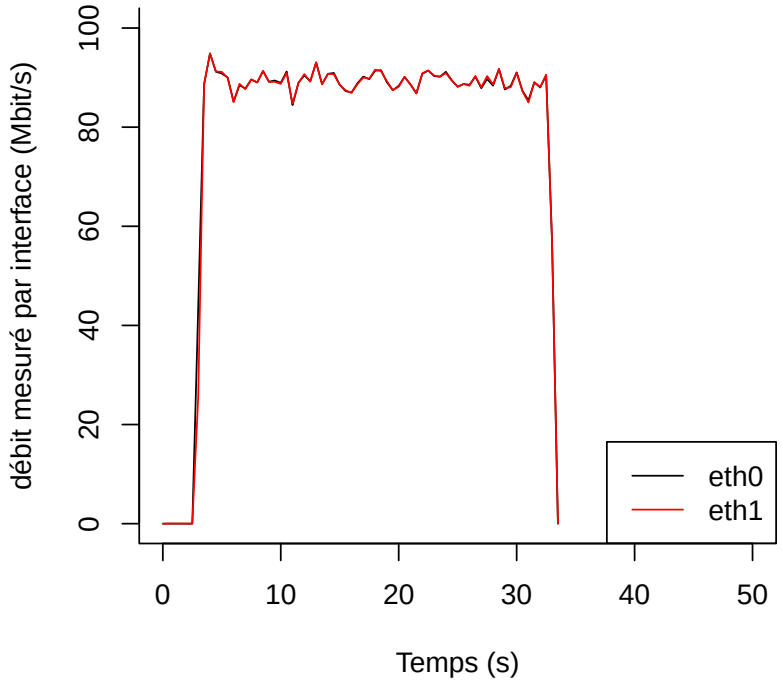


Figure 10 – **Débit entrant côté serveur.** échantillonage : 2 Hz.

et tombent en dessous des performances d'une simple connexion TCP ce qui viole la nature même de MPTCP .

Ce problème pourrait être expliqué par une utilisation non optimale de la capacité des sous-flots. Le *bandwidth delay product* (BDP) implique une taille minimale du tampon de réception. Pour un débit de 1000 Mo et un RTT de 44 ms, la taille minimale du tampon est de 5,5 Mo. Sachant que le tampon de réception [1] est partagé pour tous les sous-flots d'une connexion MPTCP, la taille minimale du tampon doit suivre cette formule [2] :

$$buffer\_size \geqslant max(\{RTT_i\}_{i\in[1,n]}) * \sum_{i\in[1,n]} Bandwidth_{\{i\}} \quad (1)$$

C'est à dire que la taille du tampon de réception doit être supérieure ou égal au produit du RTT le plus élevé parmi tous les sous-flots et la somme des capacités de tous les sous-flots. Cette taille de tampon garantit l'utilisation optimale du lien lorsque des paquets nécessitent d'être retransmisent sur des sous-flots aux délais lents. Dans notre simulation, il n'y a pas de perte de paquets, la valeur minimale des tampons correspond au BDP le

1. dans notre simulation la taille des tampons de réception est la même que la taille des tampons d'envoi.

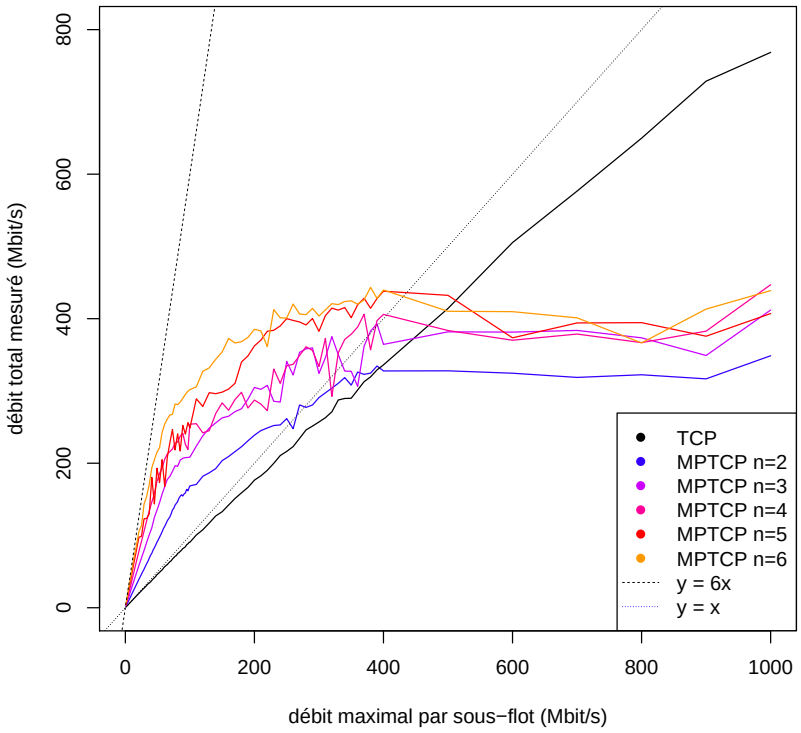


Figure 11 – **Débit total mesuré en fonction de la capacité du sous-flot.** Les débits sont mesurés avec *iperf* pour une connexion TCP classique et une connexion MPTCP contenant de 2 à 6 sous-flots.

plus élevé.

Pour les mêmes propriétés de liens, avec deux sous-flots, nous obtenons une taille minimale de 11 Mo. Mininet modife automatiquement les tampons au lancement de la topologie et les valeurs utilisées sont les suivantes (en octets) :

```
net.core.wmem_max = 16777216
net.core.wmem_default = 163840
net.core.rmem_max = 16777216
net.core.rmem_default = 163840
net.ipv4.tcp_wmem = 10240 87380 16777216
net.ipv4.tcp_rmem = 10240 87380 16777216
net.ipv4.tcp_mem = 19326 25768 38652
```

Nous observons que le tampon maximal pouvant être alloué par socket est de 16 Mo environ ce qui est largement supérieur à la taille requise. De plus en augmentant la taille du tampon, nous n'observons pas d'augmentation de performances alors qu'en le diminuant, nous observons une diminution du débit mesuré (voir Fig. 12).

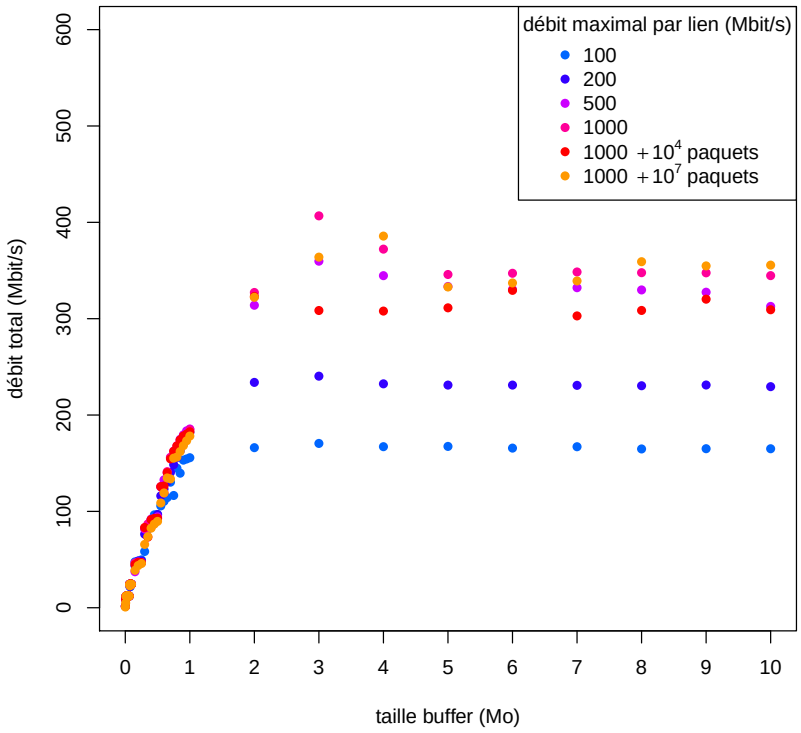


Figure 12 – **Débit mesuré en fonction de la taille de la fenêtre**. La taille maximale de la fenêtre TCP est modifiée avec l'argument '-w' avec *iperf*. La taille obtenue est étrangement deux fois supérieure à la taille demandé. Le nombre de paquets indiqué dans la légende correspond à la capacité de traitement, en nombre de paquets, des routeurs.

Nous obtenons les mêmes résultats en modifiant la capacité des routeurs ou en modifiant les valeurs de la taille des tampons dans le noyau (voir 10.1.3 page 36) que ce soit les paramètres minimum, par défaut et maxium d'*auto-tuning* de TCP (*net.ipv4*) ou les valeurs maximales ou par défaut pour tous les type de connexions (*net.core*). Une vérification de la charge CPU global avec *htop* ne montre pas une saturation des processeurs (environ 15 % d'utilisation) cependant, il reste à implémenter *cpuacct* pour vérifier la charge CPU par conteneur.

En effectuant des recherches [1] [2], il semblerait que la limite du débit est lié aux nœuds Open vSwitch sur Ubuntu 13.04, ce qui pour une dizaine de liens, limite la capacité maximale à environ 100 Mbit/s. C'est pourquoi, nous limiterons la capacité des liens à de faibles valeurs.

1. https://github.com/mininet/mininet/wiki/Introduction-to-Mininet#what-are-mininets-limitations
2. https://mailman.stanford.edu/pipermail/mininet-discuss/2014-January/003901.html

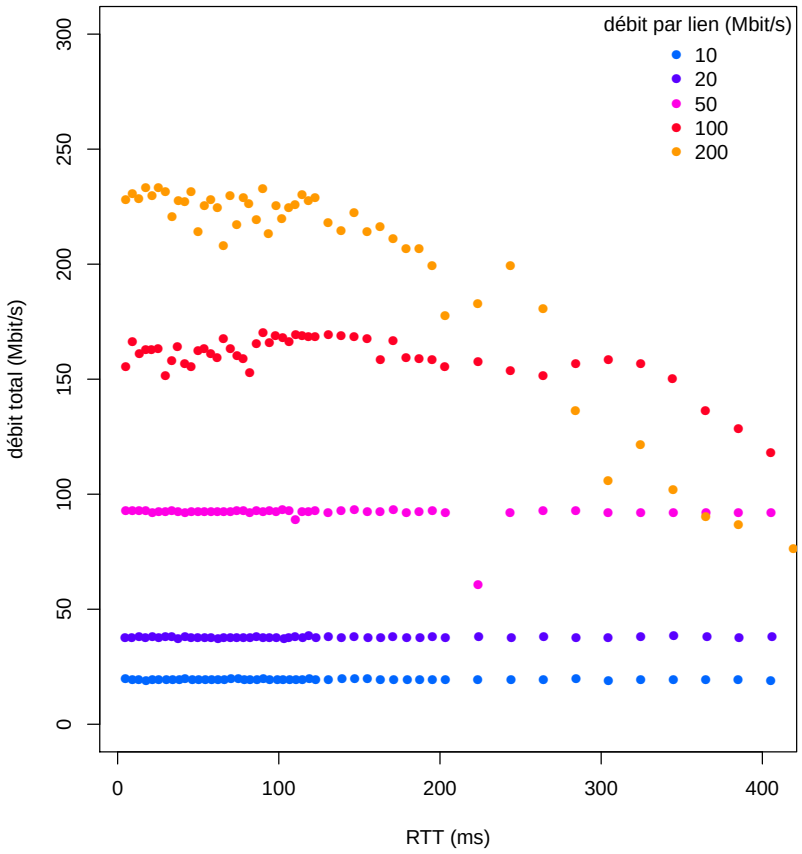


Figure 13 – **Débit maximal mesuré en fonction du délai**. Le délai est du même ordre de grandeur pour tous les liens. Le RTT est mesuré avec la commande *ping* avant le test avec *iperf* qui dure 60 s. Deux sous-flots sont utilisés.

6.3.3 Variation du délai par lien

Afin d'évaluer l'influence du délai sur le débit enregistré, le délai de tous les liens varie de manière symétrique. Le coût pour chaque sous-flot reste donc identique.

Pour mesurer le délai de chaque sous-flot, nous avons utilisé la commande *ping*. Cette approche a été utilisée car elle est la plus facile à mettre en œuvre. Par manque d'espace disque (mémoire SSD), l'utilisation de *tcpdump* est réservée pour les tests et la recherche d'erreur. Cependant, il sera nécessaire d'utiliser le délai avec TCP car cette méthode est plus fiable et l'utilisation conjointe de *ping* et d'*iperf* produit une erreur (voir 10.7.2 page 40).

Nous observons que la modification du délai n'entraîne pas de diminution des performances pour les liens à faibles capacités Fig. 13. Ce qui est attendu vu que la taille

des *buffers* et la capacité des routeurs sont largement suffisantes pour satisfaire à une augmentation de débit. Il reste alors à tester ces cas dans des conditions plus stringentes.

Pour les liens à haute capacité, l'augmentation de délai entraîne une diminution du débit total. Pour 200 Mbit/s et 400 ms de RTT, il est nécessaire d'avoir environ 10 Mo de tampon par sous-flot. Cependant, il est possible que cette diminution drastique pour le lien à 200 Mbit/s est en partie liée aux problèmes de simulation des débits élévés avec mininet.

Nous constatons que la taille du tampon s'avère être un paramètre primordial dans les performances de MPTCP. Dans une utilisation mobile, les deux sous-flots ont généralement des délais différents (par exemple, si on se base sur l'utilisation conjointe d'un accès wifi, 3G ou/et ethernet). Dans la figure 14, nous utilisons les mêmes paramètres que dans l'expérience précédente cependant le premier sous-flot aura le même RTT (une dizaine de millisecondes) pour toutes les expériences (expériences "TC"). Nous n'observons aucune différence notable entre le cas où les deux sous-flots ont un RTT variable et dans le cas où un seul des sous-flots possède un RTT variable. L'expérience "200 TC" montre que le débit est plus élevé que l'expérience "200", cette différence peut être expliquer par une taille du tampon insuffisante.

6.3.4 Choix du sous-flot en fonction du délai

Pour tester quels sont les sous-flots choisies par MPTCP dans une connexion à « faible » débit. Nous avons utilisé 5 sous-flots avec différents RTT. Le RTT de chaque sous-flot suit une sigmoïde (équation 2).

$$delay = min + \frac{max}{1 + e^{\frac{x - x_{half}}{slope}}} \tag{2}$$

Les paramètres[1] utilisés sont les suivants :

Paramètre	Valeur
delay	délai par lien
max	délai maximal par lien
min	délai minimal par lien

1. slope et x_{half} ne sont pas utiles et permettent de contraindre le nombre d'expériences et la courbure des points d'inflexion.

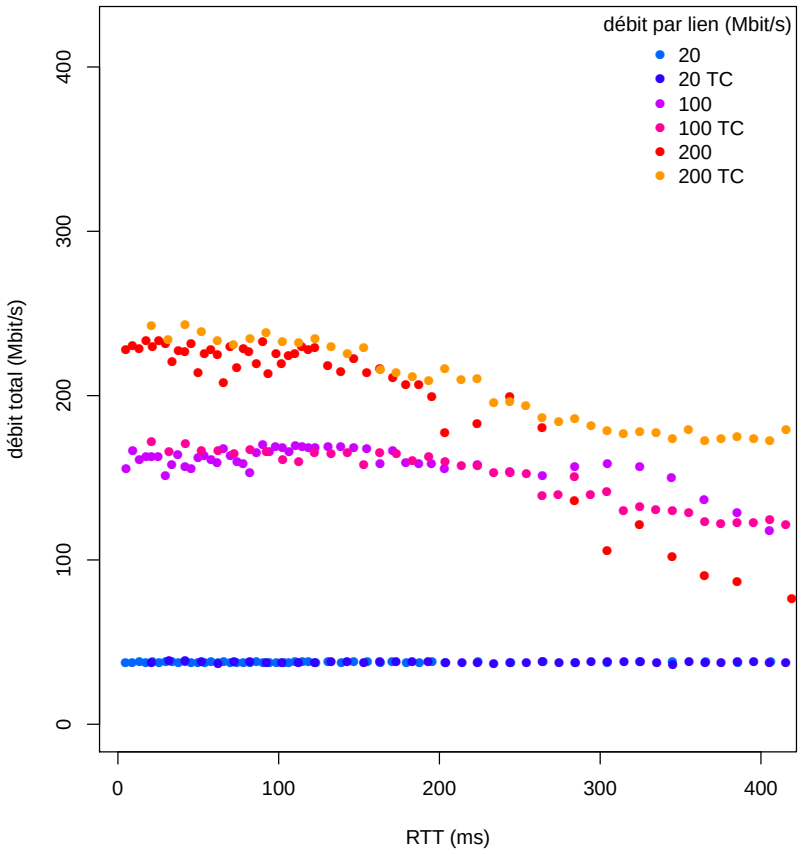
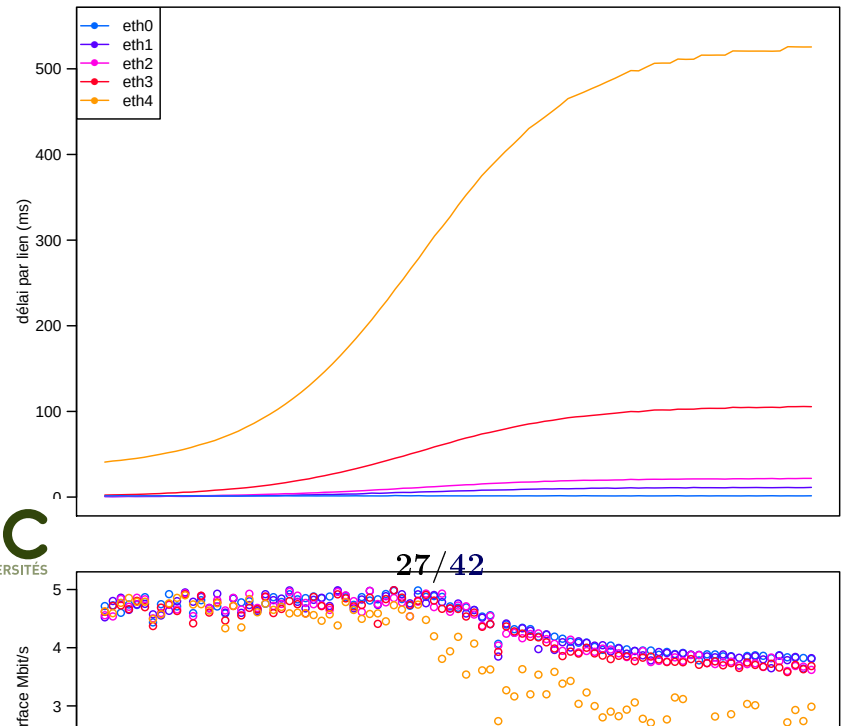


Figure 14 – **Débit total mesuré en fonction du délai d'un lien**. Le délai est modifiée seulement pour le second lien dans la topologie A (expériences "TC"). Le RTT est mesuré avec la commande *ping* avant le test avec *iperf* qui dure 60 s. Deux sous-flots sont utilisés.



Nous observons que le débit reçu est globalement équivalent pour chaque interface sauf pour l'interface "eth4" où son débit est inférieur à celles des autres pour des RTTs supérieur à 300 ms. Le débit total n'est que faiblement diminué. En effet, la commande *iperf* envoie des paquets jusqu'à atteindre la capacité de chaque sous-flot. La légère diminution des débits pour les délais long est probablement liée au tampon de réception ou d'envoi.

Il est donc difficile de prédire les sous-flots utilisés par l'ordonnanceur. *iperf* ne permet de contraindre le débit que pour des paquets UDP. Nous avons donc intégré *iperf3*[1] à la VM mininet pour permettre de fixer le débit pour des paquets TCP.

L'utilisation d'*iperf3* entraîne une erreur qui n'a pas été encore résolue à la fin de la simulation (voir 10.7.2 page 40) empêchant toutes simulations automatisées. Nous avons donc choisi deux résultats caractéristiques.

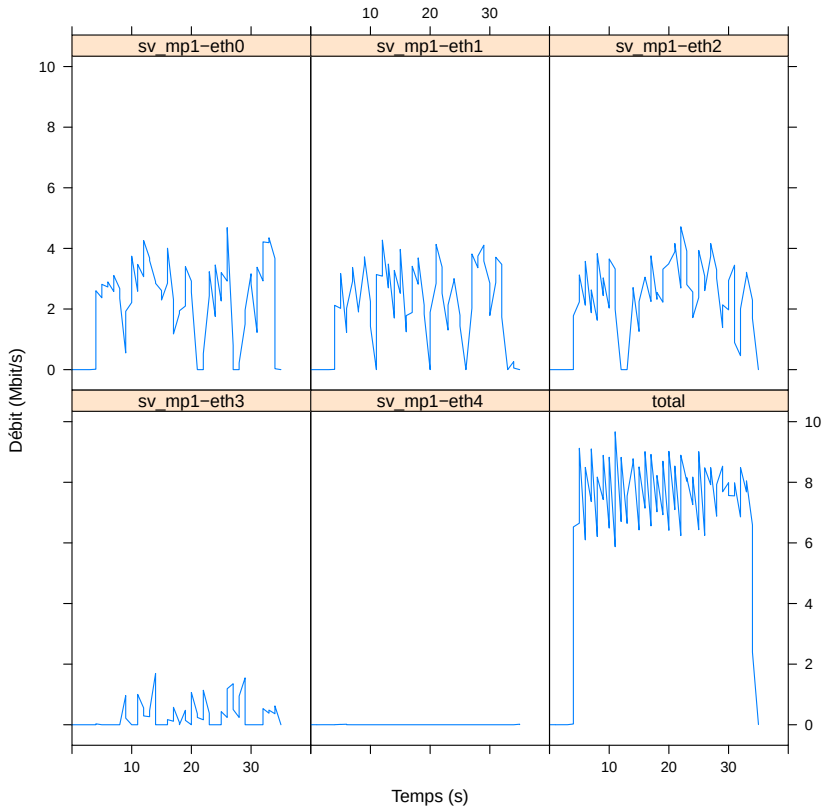


Figure 16 – **Débit mesuré pour chaque interface**. Les RTTs pour les interfaces de eth0 à eth5 sont de 34, 38, 43, 137, 552 ms. Chaque lien a une capacité de 10 Mbit/s et le débit total généré par *iperf3* a été fixé à 7,5 Mbit/s.

Dans la figure 16 et 17 nous observons que MPTCP envoie préférentiellement les

1. https://github.com/esnet/iperf

paquets dans les sous-flots à faible RTT. Il équilibre les charges dans les sous-flots où le délai reste raisonnable (sous-flot à RTT de 147 ms dans la figure 16. Cependant pour les sous-flots à « coûts » identiques, nous observons un effet de battement (*flappiness*). Il serait utile de mesurer le RTT au cours du temps pour chaque sous-flot et d'établir si cet effet de battement est lié à une variation du délai et ce pour l'algorithme LIA et OLIA [4].

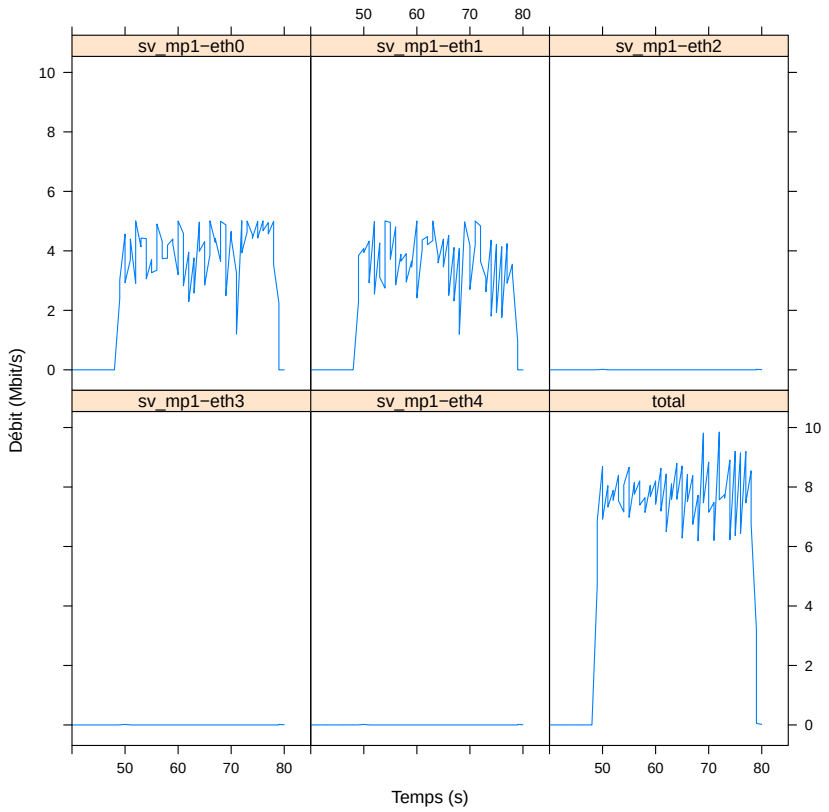


Figure 17 – **Débit mesuré pour chaque interface**. Les RTTs pour les interfaces de eth0 à eth5 sont de 43, 43, 557, 557, 557 ms. Chaque lien a une capacité de 10 Mbit/s et le débit total généré par *iperf3* a été fixé à 7,5 Mbit/s.

Une partie de ce projet consistait dans un premier temps en l'étude du code de MPTCP et de la compréhension de son fonctionnement, puis dans l'implémentation d'un nouvel ordonnanceur de notre choix. Après discussions, nous avons décidé de placer deux personnes sur l'étude du code de MPTCP car il nous semblait important d'avoir une bonne connaissance du fonctionnement du code et des principales structures pour pouvoir implémenter un nouvel algorithme d'ordonnancement.

6.4 Etude de l'algorihme de MPTCP

6.4.1 Explication de l'ordonnanceur

Lors de l'étude du code de MPTCP, nous avons étudié l'ordonnanceur déjà implémenté. L'ordonnanceur de MPTCP tient dans la fonction get_available_subflow(), qui se trouve dans ($SRC NOYAU)/net/mptcp/mptcp.output.c. Cela nous a été confirmé par Matthieu Coudron lorsqu'il a choisi l'ordonnanceur à implémenter. Cette fonction retourne la socket sur laquelle sera envoyée le prochain segment de données et définit la taille du Maximum Segment Size (MSS). Nous allons vous expliquer comment marche cet algorithme. Mais pour cela il faut connaitre d'avance les structures principales que nous allons vous expliquer avant de commenter l'ordonnanceur.
Tout d'abord la structure mptcp_cb signifie *MPTCP control block* c'est la pierre angulaire de MPTCP. Elle est utilisée dans pratiquement toutes les fonctions de MPTCP. Cette structure permet de superviser les différents sous-flots utilisés pour une connexion MPTCP. Elle permet aussi de décider s'il faut ouvrir ou fermer un sous-flot, elle réordonne les données reçues afin que l'application qui a besoin de ces données les obtienne dans le bon ordre.
La structure sock (socket) et la structure sk_buff (socket buffer) sont les mêmes que dans TCP. La socket permet de créer une liaison entre les machines : elle détient les informations nécessaires à la transmission des données ; quant au socket buffer, elle permet de définir ce que doit envoyer une socket et permet de faire un tri sur les données reçues grâce au timestamps. L'ordonnanceur implémenté dans MPTCP utilise, comme dans notre implémentation, le Smoothed Rount Trip Time (SRTT) pour déterminer quelle socket enverra les données. De ce fait, si le SRTT est petit, l'ordonnanceur utilisera plus souvent cette socket. C'est ce qui crée une faiblesse dans la sécurité de la transmission alors qu'avec MPTCP, comme il y a plusieurs chemins, la complexité peut être accrue s'il y a un équilibrage dans l'envoie des données entre les sous-flots.

Nous allons maintenant nous intéresser à la fonction get_available_subflow() qui est l'ordonnanceur.
L'algorithme se déroule de la fonction suivante :

- Tout d'abord, il y a une vérification sur le nombre de sous-flots ouverts dans la connexion MPTCP. Il est possible de récupérer cette valeur grâce à la structure mptcp_cb($mpcb- > cnt\_subflows$). On effectue ce test au préalable dans le cas où il n'y aurait qu'un seul sous-flot, au quel cas, il suffit de renvoyer l'unique socket s'il est disponible. Pour tester la disponibilité d'une socket, la fonction

mptcp_is_available() existe déjà. Elle vérifie qu'elle peut envoyer (vérification des champs de la socket), que la connexion est totalement établie, que la socket soit éligible et, que sa fenêtre d'envoi est suffisante. Si la fonction renvoie *true* alors le sous-flot peut être éligible pour envoyer des données. On peut récupérer les sockets via la structure $mpcb->connection\_list$ qui liste les sockets associées aux sous-flots.

- Sinon on regarde toutes les sockets associées à la connexion MPTCP et on distingue trois types de sockets : les backup sockets, lowpriority sockets et la meilleure socket. Dans cet algorithme, il y a des sockets qui ont une priorité basse et qui ne serviront que s'il n'existe pas de sockets avec une priorité plus grande que cette priorité. C'est ce qu'on appelle les lowpriority sockets. Parmi toutes les sockets, on choisit celle qui a le plus petit SRTT. Et on la stocke dans la variable *lowpriosk*. Bien qu'elle soit nommée lowpriority, elle a une fonction de backup mais c'est une socket éligible contrairement à la backup socket.
- Si il y a d'autres sockets qui non pas une faible priorité, on compare le SRTT de ces sockets et on garde la socket avec le RTT le plus faible. Elle est stockée dans la variable *bestsk*.
  Il y a aussi une backup socket qui sert au cas ou aucune des sockets décrites ci-dessus ne peuvent envoyer un segment de données : c'est le dernier recours car celle-ci est désignée alors qu'elle ne permet pas la réinjection de données. Elle sera stockée dans la variable *backupsk*.
- Maintenant que nous avons défini les différents types de sockets, il faut savoir sur laquelle des trois sockets on va envoyer les données ; Si il n'y a que des sockets de backup, on envoie les données sur la *lowpriosk*. Sinon on envoie les données sur la *bestsk* si elle existe. Et s'il n'y a pas de *bestsk*, on utilise *backupsk*.

6.5 Choix de l'ordonnanceur

Pour le projet, il fallait choisir et implémenter un nouvel ordonnanceur. Nous avions émis plusieurs hypothèses pour ce nouvel algorithme. En accord avec les encadrants, il a été décidé de juste utiliser le SRTT des sockets pour faire notre implémentation.

Dans un premier temps, nous avons pensé implémenter un algorithme assez simpliste mais qui permettait d'augmenter la sécurité contre les attaques de type *Man In The Middle*. C'est à dire que notre algorithme allait envoyer les segments de manière équitable sur chaque sous-flots permettant l'envoi de données. Cet algorithme avait pour vocation de rendre plus difficile la récupération d'informations en écoutant un sous-flot car avec l'algorithme actuel, si un attaquant voulait récupérer un maximum d'informations, il lui suffisait d'écouter le sous-flot qui a le SRTT le plus faible. C'est ce que nous voulions éviter avec notre algorithme. Cependant après de plus amples réflexions, nous avons remarqué que notre algorithme avait un grand nombre de défauts.
En effet, MPTCP a été développé afin d'avoir une amélioration des débits mais cet algorithme aurai empêché cela si le débit d'un sous-flot était vraiment faible comparé aux

autres sous-flots alors toute la connexion MPTCP ressentirait ce débit faible et cela influerai beaucoup sur les performances de MPTCP. C'est pourquoi cet algorithme n'a pas été choisi.

Un compromis entre sécurité et performances est nécessaire pour que l'utilité de MPTCP soit avantageuse par rapport à TCP. Il nous a été proposé de sélectionner les k meilleurs sous-flots d'un point de vue du SRTT et de faire un Round-Robin sur ces k sous-flots. k étant laissé à notre appréciation. Cet algorithme est le meilleur compromis trouvé car il permet de ne pas envoyer de données sur un sous-flot si son SRTT est trop grand, ce qui permet de garder une certaine rapidité dans la transmission de données et de garder une certaine sécurité car le trafic passe de manière équitable sur les k meilleurs sous-flots.

6.6 Implémentation de l'ordonnanceur

Afin de pouvoir stocker les k meilleurs sous-flots, on avait 2 choix.

- Soit un tableau.
- Soit une liste chainée.

Après en avoir discuté entre nous, nous avons décidé d'utiliser une liste chainée car les listes chainées permettent une plus grande flexibilité de manipulation. Nous avons donc déclaré dans *mptcp.h*, une structure :

```
struct selected_sk{
  struct sock *sk;
  struct selected_sk *next;
};
```

Cette structure permet de pointer sur une socket et de d'avoir un lien vers la socket suivante ce qui permet de naviguer très facilement entre les différentes sockets sélectionnées. Ce qui est très utile pour le Round-Robin.

Une fois que l'on avait déclaré cette structure, il fallait aussi implémenter des fonctions afin de pouvoir faire une liste chainée en fonction des RTT. Notre liste chainée sera organisée de telle sorte :
On aura une *selected_sk* qui sera appelée *bssk*. C'est la "best selected socket". Ca sera la socket sélectionnée qui aura le meilleur RTT, puis son attribut next pointera sur la deuxième sockets avec le meilleur RTT. Et la k^{eme} socket aura pour *next* la *bssk*. Cela formera une boucle. Les fonctions que l'on a créés sont :

- *static void ssk_insertion_sort(struct selected_sk *bssk, int ssk_size);* : qui permet de faire en sorte que la liste chainée soit triée en fonction du Smoothed RTT (SRTT) des sockets.
- *static u32 ssk_max_srtt(struct selected_sk *bssk);* : permet de retourner la valeur maximale du SRTT de la liste chainée, c'est à dire le RTT de la socket qui a pour*next* la bssk passée en argument.

- *static int belongto_ssk(struct sock *sk, struct selected_sk *bssk, int ssk_size);* : permet de savoir si la socket sk passée en argument appartient à la liste chainée.
- *static struct selected_sk *bssk_prev(struct selected_sk *bssk);* : permet d'obtenir la socket ayant le plus grand SRTT de la liste chainée. C'est la socket précédent la *bssk*.
- *static void ssk_checkup(struct sk_buff *skb, struct selected_sk *bssk, int ssk_size);* : permet de retirer les sockets de la liste chainée si elles ne sont pas capable d'envoyer des données ($!mptcp\_is\_available(it->sk, skb, \&this\_mss)$) et si on a déjà mis dans la queue de la socket le buffer *skb* ($mptcp\_dont\_reinject\_skb(tp, skb)$).

Où *ssk_size* est le nombre de socket qui forme la liste chainée.

Avec ces fonctions nous pouvons créer la fonction principal que l'on placera dans *static struct sock *get_available_subflow(struct sock *meta_sk, struct sk_buff *skb, unsigned int *mss_now);*
Nous allons maintenant expliquer comment fonctionne la fonction principale : On vérifie tout d'abord si tous les sockets de la liste chainée ont envoyé une fois (si elles existent toujours) et si oui, on recalcule la liste chainée. Dans cette boucle qui permet d'établir cette liste chainée, on teste chaque socket si elle est disponible. Si la socket a un meilleur SRTT ou qu'il reste de la place dans les k meilleures sockets, on rajoute cette socket dans la liste. Pour cela, une insertion est effectuée en gardant la liste triée. Par contre, si le SRTT est plus grand que ceux de la liste chainée et qu'il y a déjà k sockets, on passe à la socket suivante.

7 Conclusions

Au cours de ce projet, nous avons

- compilé le noyau MPTCP dans une machine virtuelle contenant mininet
- créé une plateforme de tests par des scripts python et bash
- testé le fonctionnement de MPTCP sur une topologie simple et sur la topologie *fat-tree*
- testé les capacités et les limites de la virtualisation de réseaux
- puis mesuré les performances de MPTCP sur des liens à délai variables
- enfin, nous avons écrit un nouvel algorithme d'ordonnancement
- et nous avons compilé le noyau avec ce nouvel algorithme

8 Perspectives

Le sujet est très intéressant et les pistes pour continuer le projet sont nombreuses.

Dans un premier temps, il serait profitable de terminer les scripts de base en intégrant des moyens pour mesurer la charge CPU par conteneur (*cpuacct*) ou de mesurer le RTT par la connexion TCP. Ensuite, il serait utile de faire varier d'autres paramètres comme la probabilité d'erreur, la gigue sur chacun des liens pour ensuite, effectuer des expériences sur des liens congestionnés où plusieurs utilisateurs sont en concurrence pour la ressource. Ces tests stringents seront utiles pour mesurer la performance des algorithmes contenus dans le noyau et pour les comparer avec le nouvel algorithme.

9 Remerciements

Ce projet a été réalisé dans le cadre du Master I Réseau, à l'Université Pierre et Marie Curie.

Nous tenons à remercier Stefano Secci pour nous avoir permis de réaliser ce projet.

Et nous remercions Yacine Benchaïb et Matthieu Coudron pour les discussions autour du projet.

10 Annexe1

10.1 Utilisation des scripts pythons

10.1.1 Activation MPTCP

Il y a deux fichiers qui nous intéressent :

```
/proc/sys/net/mptcp_path_manager
/proc/sys/net/mptcp_enabled
```

Le fichier enabled permet d'activer (*1*) ou de désactiver (*0*) MPTCP sur la machine.

Le fichier path_manager influence sur la méthode d'annonce des sous-flots. Deux valeurs nous intéressent : *fullmesh*, et *default*. La valeur *default*, il n'y a pas d'annonces des sous-flots disponibles tandis que *fullmesh* permet la création d'un *mesh* de sous-flots parmi tous ceux disponibles.

Pour modifier la valeur du fichier, il suffit d'utiliser sysctl en utilisant la forme "nom=valeur"

```
sudo sysctl -w net.mptcp.mptcp_path_manager=fullmesh
sudo sysctl -w net.mptcp.mptcp_enabled=1
```

Attention, selon les choix lors de la compilation du noyau, la valeur par défaut du path_manager peut être mise sur "default".

10.1.2 Choix de l'algorithme congestion

Pour choisir l'algorithme de congestion, il est nécessaire de modifier le fichier suivant :

```
/proc/sys/net/ipv4/tcp_congestion_control
```

Pour l'algorithme LIA :

```
sudo sysctl -w net.ipv4.tcp_congestion_control=coupled
```

Pour l'algorithme OLIA :

```
sudo sysctl -w net.ipv4.tcp_congestion_control=olia
```

Pour l'algorithme par défaut :

```
sudo sysctl -w net.ipv4.tcp_congestion_control=cubic
```

Pour afficher les algorithmes disponibles :

```
sudo sysctl net.ipv4.tcp_available_congestion_control
```

L'ordonnanceur par défaut est choisi à la compilation du noyau.

10.1.3 Taille de la fenêtre

Pour modifier la taille maximale de la fenêtre de réception (*rmem*) et d'envoi *wmem*, il faut modifier les fichiers suivants :

```
/proc/sys/net/core/wmem_max
/proc/sys/net/core/rmem_max
/proc/sys/net/core/rmem_default
/proc/sys/net/core/rmem_default
```

Pour une taille *buffer* maximale de ∼16 Mo :

```
sysctl -w net.core.wmem_max=16777216
sysctl -w net.core.rmem_max=16777216
```

Pour modifier les valeurs par défauts :

```
sysctl -w net.core.rmem_default=163840
sysctl -w net.core.wmem_default=163840
```

Pour modifier les valeurs d'*auto-tuning* :

```
sysctl -w net.ipv4.tcp_wmem='4096 16384 4194304'
sysctl -w net.ipv4.tcp_rmem='4096 87380 6291456'
```

Les valeurs correspondent respectivement à la taille minimale, par défaut et maximale du buffer alloué (en octets) par socket TCP. La taille maximale ne peut dépasser celle indiquée dans *net.core*.

10.2 Lancement scripts python

Les fichiers permettant de lancer la topologie et d'exécuter les tests se nomment en commençant par "pyMPTCP" et se situent sur le git pour l'instant dans "./mininet/python/TCPvsMPTC

```
pyMPTCP.py
pyMPTCP_parser.py
pyMPTCP_topo.py
pyMPTCP_options.py
```

pyMPTCP.py contient le *main* et doit être modifié pour choisir la topologie (objet *topo* dans la fonction *main* et objet *net* dans la fonction *runMPTCP*).

pyMPTCP_topo.py contient les définitions des topologies utilisées. La classe *Topo* sera utilisée par mininet pour la création du réseau. La classe *Topo->names* contient les noms des hôtes et des switches qui seront utiles pour les tests.

pyMPTCP_parser.py contient le parseur d'argument voir section 10.4.

pyMPTCP_options.py contient les fonctions qui seront lancées selon les arguments utilisés.

La fonction options permet de modifier la topologie après la création des nœuds. À cause d'un bug de mininet ,pour l'instant encore non résolu, il est nécessaire d'utiliser la classe *mininet* pour pouvoir créer plus d'un lien entre deux mêmes nœuds.

Exemple d'utilisation :

```
sudo python ./pyMPTCP.py -O exp001_TC --bw 10 --mptcp -n 5 -t 60 --shark
```

Ici nous lançons la topologie A, avec l'expérience exp001_TC avec un débit maximal par lien de 10 Mbit/s, en utilisant MPTCP, avec 5 sous-flots et *iperf* sera utilisé pendant 60 secondes, et tcpdump sera utilisé pour enregistrer les paquets. Pour plus de précisions voir 10.4.

10.3 Résumé des arguments pour le parseur

Les commentaires des arguments sont disponibles dans la section 10.4.

```
usage: pyMPTCP.py [-h] [--verbose] [--tc] [--sshd] [--cli] [--csv] [--dump]
                  [--shark] [--bwm_ng] [--output OUTPUT] [--prepend PREPEND]
                  [--postpend POSTPEND] --open OPEN [--file FILE] --bw BW
                  [--delay DELAY] [-n N] [-t T] [--maxq MAXQ] [--mptcp]
                  [--pause] [--ndiffports NDIFFPORTS] [--arg1 ARG1]
                  [--arg2 ARG2] [--arg3 ARG3] [--arg4 ARG4] [--arg5 ARG5]
```

Description

--mptcp	active mptcp
--bwm	fixe la capacité maximale de **tous** les liens
--delay	fixe le délai (direction symétrique) de **tous** les liens
--bwm-ng	lance bwm-ng sur le cient et le serveur
--shark	lance tcpdump sur toutes les interfaces client et serveur et écrit la sortie dans un fichier "[hôte]-[if].pcap"
--maxq n	taille maxiamle de la queue au niveau des *switchs*
--open	fichier "expérience" utilisé (contenu dans le dossier "./experiment"
--output	dossier contenant les fichiers de sortie
--file	nom des fichiers de sortie
--prepend	préfixe pour les noms de fichiers
--postpend	suffixe pour les noms de fichiers
--cli	lance le mode *command line interface* avant le lancement de l'expérience
--csv	la sortie de la commande iperf est formatée en csv
--delay	fixe le délai de tous les liens
--pause	pause après la simulation
--sshd	lance sshd sur chaque hôte. Connexion du réseau à l'espace de nom de la racine.
--t n	*iperf* durée en seconde de l'expérience
--n n	nombre de sous-flots
--tc	active la modification des liens via *tc*
-argx	en conjonction avec -- tc, permet de rentrer d'autres arguments avec les fichiers *experiment*
--ndiffports n	non utilisé
--verbose	non utilisé
--dump	non utilisé

exemple :

```
sudo mptpc_khal.py --cli --delay "50ms"
```

10.4 arguments mininet

```
usage: sudo mptc_khal.py [h] [--cli] --bw BW [--delay DELAY] [-n N] [-t T]
                         [--mptcp] [--pause] [--ndiffports NDIFFPORTS]
```

cf argument parser annexe.

10.4.1 ssh

```
usage: sudo mptc_khal.py --sshd
```

Cette option lance le démon ssh dans chacun des nœuds avec la commande

```
/usr/sbin/sshd -o UseDNS=no -u0
```

Pour l'instant, seul la connexion à partir de root avec le premier hôte (172.16.0.1) fonctionne. En effet la résolution ARP échoue pour les autres hôtes (pas de paquets ARP reply en utilisant tcpdump).

sur le nœud root

```
tcpdump -i root-eth0 arp
```

Cependant, il reste possible de se connecter aux autres hôtes par ssh via le premier hôte.

Pour l'instant, une méthode « sale » est utilisée pour arrêter les démons sshd sur les hôtes.

10.5 scripts shell

Les scripts écrits en bash permettent de lancer plusieurs fois les simulation en utilisant des paramètres variables. Il est nécessaire de les lancer en mode super-utilisateur.

```
sudo bash shNAME.sh
```

shKernelCheck.sh	Permet de visualiser l'algorithme de congestion et les taille des tampons
shKernelDefault.sh	Permet de modifier les tailles des buffers
shMPTCP.sh	lanceurs d'expériences

Le script shMPTCP permet de multiplier les expériences. Les fonctions écrites ne sont pas toutes utilisées cependant il y a 2 fonctions importantes qu'on peut décrire.

clean

```
sudo bash shMPTCP.sh clean
```

Cette fonction envoie un signal d'arrêt aux fonctions qui sont utilisées pour les mesures : (ping, iperf, iperf3, bwm-ng, tcpdump), elle supprime tous les fichiers de sortie situés dans ./output/ et effectuer un nettoyage de "mininet" (mn -c) en supprimant les noeuds, liens hôtes créés.

runMPTCP Cette fonction exécute le script python est appelé à l'intérieur d'une boucle par d'autres fonctions.

Les autres fonctions appellent la fonction runMPTCP pour effectuer les simulations, exemple :

```
sudo bash shMPTCP.sh runbw
```

10.6 Scripts R

Les scripts R ne sont pas totalement finalisés pour être exploitables facilement par autrui et pour certains, il faudra modifier les fichiers pour indiquer les dossiers contenant les résultats des expériences.

Le dossier ./scripts/R contient les fonctions qui permettent d'analyser les fichiers produits par les scripts pythons.

Le dossier ./scripts/Analysis contient les fichiers expérimentaux et les fichiers R permettant de les analyser en faisant appel aux fonctions se trouvant dans le dossier ./scripts/R. Pour une question de stockage, les fichiers des différentes expériences ne sont pas sur le git.

10.7 Bugs

10.7.1 Topologie

À la création de la topologie mininet, il est nécessaire de donner des noms courts aux hôtes et aux switchs.

Il n'est pas possible de créer deux liens entre les mêmes nœuds de la classe "Topo". Pour contourner cette limitation, il faut créé la topologie avec un seul lien puis dans la classe "Mininet", rajouter le second lien. C'est pourquoi, dans le fichier "*pyMPTCP_topo.py*", il y a deux définitions pour la même topologie : une pour la classe Topo (*topo()*) et une autre pour la classe Mininet (*topo_options(args,net)*) qui sera executé séquentiellement lors de la simulation.

10.7.2 mininet

iperf et ping L'utilisation conjointe de la commande *iperf* et de la commande ping dans une simulation produit une erreur du calcul du délai par la commande ping. Ce délai augmente exponentiellement vers environ 1000 ms. Si on analyse les traces enregistrés avec *tcpdump*, les réponses aux *echo request* se produisent bien plus rapidement que ne laissent sugérer les résultats de la commande *ping*. L'utilisation conjointe de ces deux applications marchent très bien entre la machine physique et la VM et il est probable que ce problème soit lié à mininet.

sshd

```
usage: sudo mptc_khal.py --sshd
```

Cette option lance le démon ssh dans chacun des nœuds avec la commande

```
/usr/sbin/sshd -o UseDNS=no -u0
```

Pour l'instant, seul la connexion à partir de root avec le premier hôte (172.16.0.1) fonctionne. En effet la résolution ARP échoue pour les autres hôtes (pas de paquets ARP reply en utilisant tcpdump).

sur le nœud root

```
tcpdump -i root-eth0 arp
```

Cependant, il reste possible de se connecter aux autres hôtes par ssh via le premier hôte.

MSS Lorsque nous fixons le *Maximum Segment Size* à 1460 octets, nous obtenons ce message d'erreur :

```
WARNING: attempt to set TCP maximum segment size to 1460, but got
536
```

Cependant, si nous analysons les paquets enregistrés grâce à *tcpdump*, nous observons que le MSS négocié pendant le *handshake* de la connexion est bien de 1460 octets et que la taille des paquets de données est de 1428 octets.

iperf3 Dans l'état des scripts, *Iperf3* produit une erreur à la fin de la simulation. Il est nécessaire de nettoyer la simulation et il n'est pas possible pour l'instant d'utiliser les scripts *shell* pour effectuer des simulations automatisées.

```
sudo bash shMPTCP.sh clean
```

Références

[1] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, "Architectural guidelines for multipath tcp development," *RFC 6182*, March 2011. (Cité en pages 3, 4, 10 et 21.)

[2] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "Tcp extensions for multipath operation with multiple addresses," *RFC 6824*, January 2013. (Cité en pages 3 et 22.)

[3] M. Coudron, S. Secci, G. Pujolle, P. Raad, and P. Gallard, "Cross-layer cooperation to boost multipath tcp performance in cloud networks," in *Cloud Networking (CloudNet), 2013 IEEE 2nd International Conference on*, pp. 58–66, IEEE, 2013. (Cité en pages 3 et 11.)

[4] R. Khalili, N. Gast, M. Popovic, U. Upadhya, and J.-Y. Le Boudec, "Mptcp is not pareto-optimal : Performance issues and a possible solution," *Networking, IEEE/ACM Transactions on*, vol. 21, no. 5, pp. 1651–1665, 2013. (Cité en pages 3, 4, 11, 13, 14, 17 et 28.)

[5] R. Stewart, "Stream control transmission protocol," *RFC 4960*, September 2007. (Cité en page 9.)

[6] D. Thaler, Microsoft, C. Hopps, and N. Technologies, "Multipath issues in unicast and multicast next-hop selection," *RFC 2991*, November 2000. (Cité en page 9.)

[7] C. Paasch and O. Bonaventure, "Multipath tcp," *Communications of the ACM*, vol. 57, pp. 51–57, April 2014. See http ://queue.acm.org/detail.cfm ?id=2591369. (Cité en page 10.)

[8] C. Raiciu, M. Handly, and D. Wischik, "Coupled congestion control for multipath transport protocols," *RFC 6356*, October 2011. (Cité en pages 10 et 20.)

[9] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, and M. Handley, M.ndley, "Data center networking with multipath tcp," in *Proceedings of the 9th ACM SIGCOMM*, pp. 58–66, IEEE, 2010. (Cité en page 11.)