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

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 s ur 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 subdivisé 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'optimum

1. mptcp.org

de pareto par rapport aux objectifs de MPTCP, 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 virtualisées et de tester les performances de MTPCP en faisant varier les paramètres des sous-flots. La seconde partie est de construire un alogrithme d'ordonnancement répondant à des critères de sécurité.

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 : *fat tree* pour simuler un *data center* et une topologie permettant de tester la concurrence entre MPTCP et TCP ;
- 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 des différents algorithmes.

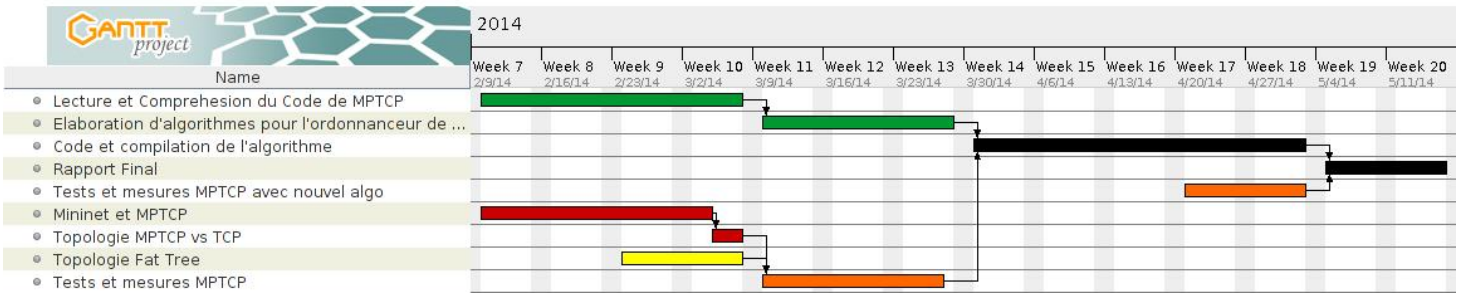

Figure 1 – **Diagramme de Gantt général**. Les couleurs correspondent à la répartition entre les membres de l'équipe : en *rouge* M. Ly, en *jaune* M. Ravier et en *vert* M. Dubois et M. Lam, en *orange* M. Ravier et M. Ly et en *noir* par tout le monde.

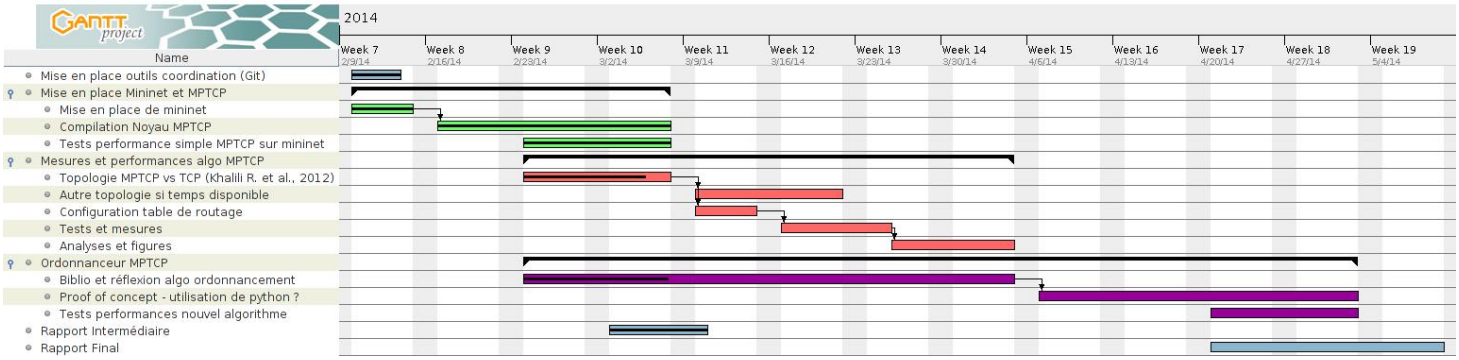

Figure 2 – **Diagramme de Gantt personnalisée Romain Ly.**

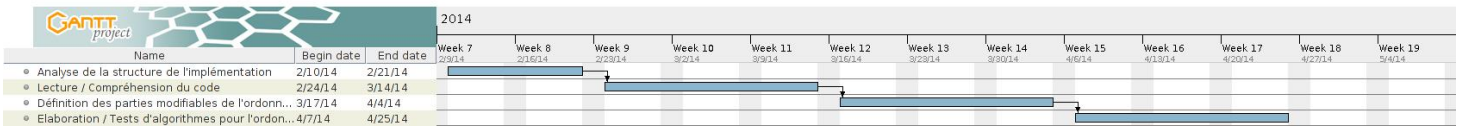

Figure 3 – **Diagramme de Gantt personnalisée Kevin Lam et Quentin Dubois.**

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 et permet de multiplexer les flux sur plusieurs interfaces [5]. ECMP (*Equal Cost MultiPath*) est un protocole qui semblait prometteur dans les data center. Lors d'une connexion entre deux hôtes, le routeur peut transférer les paquets sur plusieurs meilleurs chemins à coûts « égaux » [6]. L'inconvénient 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. ECMP nécessite le travail des routeurs pour connaître les chemins et l'augmentation de performance n'est pas forcément significative. L'avantage de MPTCP est d'être transparent par rapport à TCP, c'est à dire que si un hôte n'est pas compatible avec MPTCP, la connexion retournera vers une connexion TCP classique. L'autre 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 [1]

- le gestionnaire de chemin ou *path manager*
- l'ordonnanceur de paquets (*packet scheduling*)
- le contrôleur de congestion

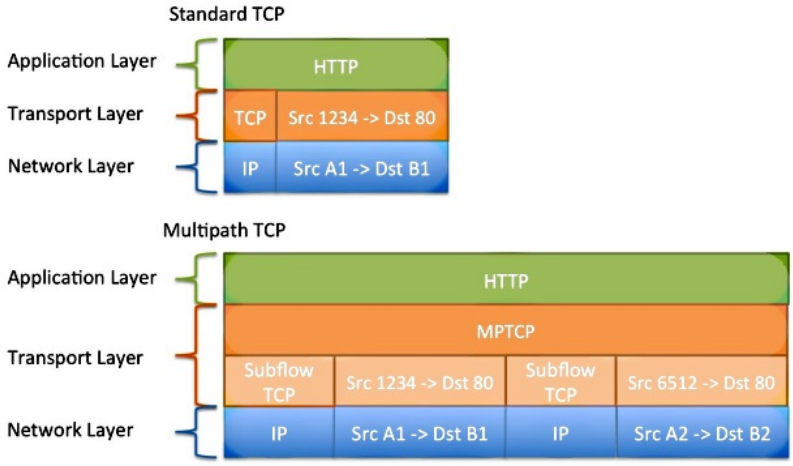

Figure 4 – **Schéma de la pile MPTCP**. source : www.cisco.org

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 [7].

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. 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 [8]. 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.

Par exemple, l'utilisation de chiffrement de type CBC (*Cipher Block Chaining*) compliquera l'attaquant car il sera nécessaire d'obtenir le bloc n-1 pour déchiffrer le bloc n. AES-CBC est utilisé couramment dans des communications de type HTTPS/SSL. L'attaquant devra disposer de tous les paquets sans exception pour pouvoir déchiffrer le message en supposant qu'il possède la clé adéquate.

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. 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.

4 Analyse

La partie Analyse sera étoffée dans le rapport final par l'analyse des résultats obtenus.

5 Conception

5.1 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. 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.

5.1.1 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.

5.1.2 FatTree

Afin de tester MPTCP de manière réaliste, nous avons simulé une topologie FatTree, souvent utilisée dans les datacenters qui sont les premiers nécessiteux 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.

5.1.3 MPTCP vs TCP

Pour déterminer les critères à respecter de l'ordonnanceur (celui par défaut, ou l'O-LIA) 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).

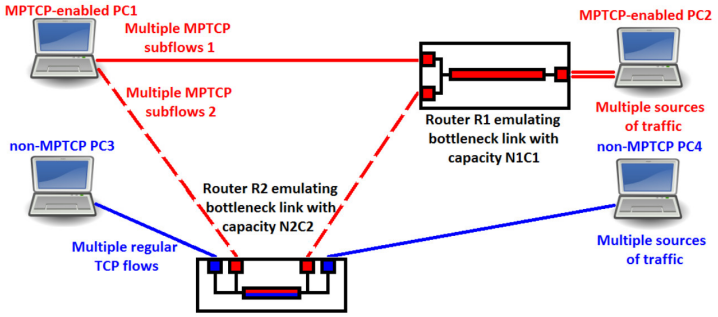

Figure 5 – **Testbed MPTCP vs TCP [4]**.

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).

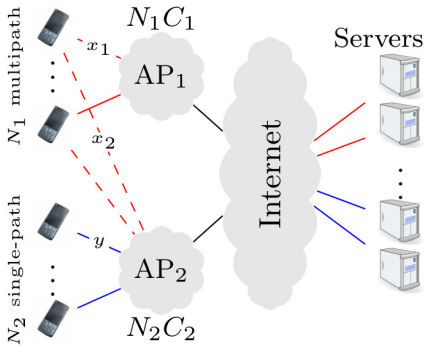

Figure 6 – **Testbed MPTCP vs TCP [4]**. Les N1 utilisateurs MPTCP (rouge) utilisent deux points d'accès pour se connecter à un serveur distant dont un qui est partagé avec les N2 utilisateurs TCP (bleu).

5.2 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'erreurs, la gigue, etc. Nous testerons quelle est l'influence de ces paramètres sur le choix des sous-flots par l'ordonnanceur.

5.3 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 affecté. 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 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 pourrait être variable selon d'autres paramètres comme le nombre de sous-flots disponibles.

5.4 Test de l'algorithme d'ordonnancement

L'écriture et le test de l'algorithme d'ordonnancement dans le noyau linux peut s'avérer une tâche difficile en si peu de temps. Pour tester la validité de notre algorithme d'ordonnancement, nous réfléchissons à effectuer d'abord un *proof of concept* en utilisant directement python qui utilisera des fonctions de *callback* pour certaines fonctions du noyau nécessaire à MPTCP. On utilisera alors UDP pour la transmission des données.

6 État d'avancement

6.1 Outil de coordination : git

L'état des scripts utilisées par l'équipe est mise à jour par un système de version utilisant git `https://github.com/Romain-Ly/PRES`.

6.2 Compilation MPTCP [1]

La mise en place du noyau linux MPTCP (v0.88) dans une VM de mininet (v2.10) est à 100 % terminé.

Les paquets debian pour l'installation du noyau MTPCP sur une VM de mininet est disponible : (`https://www.dropbox.com/sh/y4ykck8rg6908ps/7V3lsV6Ggg`).

Pour tester la réussite de l'installation, une topologie deux hôtes et n switchs a été utilisé. L'utilisation de MPTCP montre un débit supérieur lorsque l'on compare à la même expérience où MPTCP a été désactivé dans le noyau.

6.3 FatTree [2]

Chargé de la conception du réseau de test, ma première préoccupation a été de maîtriser Mininet. Après documentation, je me suis penché sur l'API Python offert par cet outil. Après cette prise en main concrétisée par quelques tests de connectivité sur des topologies simples, j'ai commencé à coder ma propre topologie, un FatTree à 2 niveaux avec des switches à 36 ports. N'ayant pas trouvé de définition formelle du FatTree, je me suis contenté d'une instance particulière, relativement simple mais permettant tout de même à MPTCP d'emprunter plusieurs sous-flots différents pour se rendre d'un hôte à l'autre. Suite au travail de Romain, la prochaine tâche sera d'installer le noyau MPTCP sur la machine virtuelle Mininet, de le configurer, puis faire en sorte qu'il soit correctement utilisé dans notre réseau FatTree. J'effectuerai ensuite plusieurs tests de performance sur cette topologie, probablement en collaboration avec Romain.

6.4 Topologies virtualisées [3]

J'ai reproduit la topologie où MPTCP est en concurrence avec un flux TCP [4]. Il reste à établir les tables de routage de chaque hôte pour pouvoir tester les performances de MPTCP.

1. par M. Ly
2. par M. Ravier
3. par M. Ly

6.5 Code MPTCP [1]

Nous avons regardé les fichiers de MPTCP pour avoir une vision globale de l'implémentation dans le noyau linux et essayer de déterminer les fichiers qui concernent l'ordonnancement des sous-flux. Nous avons ensuite essayé de déterminer où nous pouvions modifier le code afin d'adapter l'ordonnanceur aux besoins du projet. Nous avons avancé sur cette phase de compréhension du code mais il nous reste toujours à savoir où nous pouvons modifier le code sans rendre MPTCP non fonctionnel ou non performant. Pour cela, il faudra tester sur des topologies virtuelles simples et comparer les différences de performances. Bien sûr, dans les tests nous ne codons que des ordonnanceurs idiots : ils effectueront uniquement une répartition équitable des sous-flux sachant qu'ils ont tous le même débit.

1. par M. Lam et M. Dubois

7 Compte rendu du projet

7.1 Outil de coordination : git

L'état des scripts utilisées par l'équipe est mise à jour par un système de version utilisant git https://github.com/Romain-Ly/PRES.

7.2 Compilation MPTCP

Pour la réalisation du projet, nous utilisons une machine virtuelle de mininet installé ubuntu 13.04 32 bits, disponible ici : https://bitbucket.org/mininet/mininet-vm-images/downloads.

La mise en place du noyau linux MPTCP (v0.88) dans une VM de mininet (v2.10) est à 100 % terminé.

Les paquets debian pour l'installation du noyau MTPCP sur une VM de mininet est disponible : (https://www.dropbox.com/sh/y4ykck8rg6908ps/7V3lsV6Ggg).

7.3 Topologie

Pour effectuer les expériences de débit avec MPTCP, nous utilisons principalement deux topologies (A et B).

La topologie A est inspiré du *testbed* de l'article de R. Khalili [4], voir Fig. 7. Nous avons ajouté à cette topolgoie des routeurs privés entre le client et le serveur *MPTCP* pour augmenter le nombre de sous-flots.

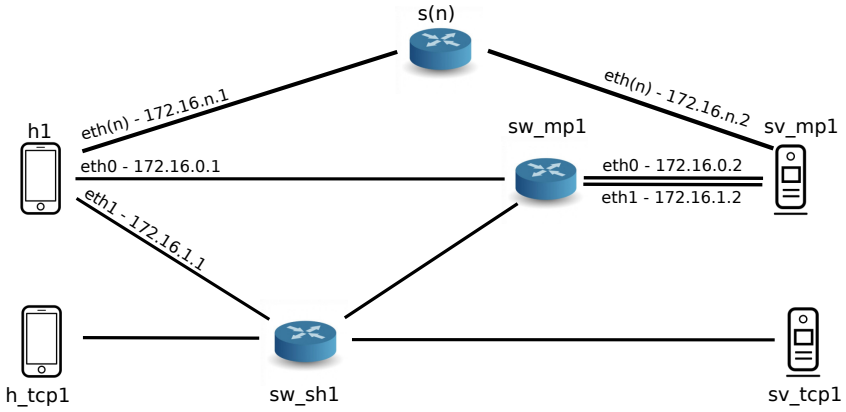

Figure 7 – **Reproduction de la topologie de l'article de R. Khalili**. Le(s) *switch(s)* n ne sont présent(s) que si le nombre de sous-flot est supérieure à deux. Pour n sous-flots, il y aura création de n-2 *switchs* et autant de liens supplémentaires. L'hyperviseur est connecté à tous les switchs.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

7.4 Performance de MPTCP sur mininet

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

Les paramètres [1] utilisés sont les suivants :

Paramètre	Valeur
MSS	1460 octets
window size	85,3 Koctets
délai par lien	10 ms
Algorithme de congestion	LIA [7]

Si nous fixons le MSS 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 la taille maximal du MSS négocié pendant le *handshake* de la connexion MPTCP est de 1460 octets et que la taille du MSS dans les paquets de données est de 1428 octets.

7.4.1 Un exemple de démonstration

Ici nous allons prendre l'exemple d'une connexion avec deux sous-flots à 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 8.1 page 25 et 8.3 page 27.

Le délai (RTT) entre h1 et sv_mp1 est de 44 ± 11 ms (mesuré avec la commande ping), ce qui correspond bien au délai nécessaire pour traverser deux liens. La moyenne ici tient compte du délai du premier paquet qui est envoyé vers le contrôleur qui va établir le chemin vers le serveur [2].

L'argument "--bwm-ng" permet de lancer Bandwidth Monitor NG [3] (bwm-ng) et de mesurer plusieurs paramètres comme le nombre d'octets, de paquets, le débit entrant ou sortant passant par chacune des interfaces sondées. La Figure 8 montre le débit entrant côté serveur pour ses deux interfaces, nous observons que pour deux sous-flots aux capacités identiques, le débit mesuré est quasi identique (une différence de quelques paquets est observée).

1. paramètres par défaut dans le noyau Linux
2. Il pourrait être nécessaire d'enlever le délai de ce premier paquet dans une version future.
3. http://www.gropp.org/?id=projects&sub=bwm-ng

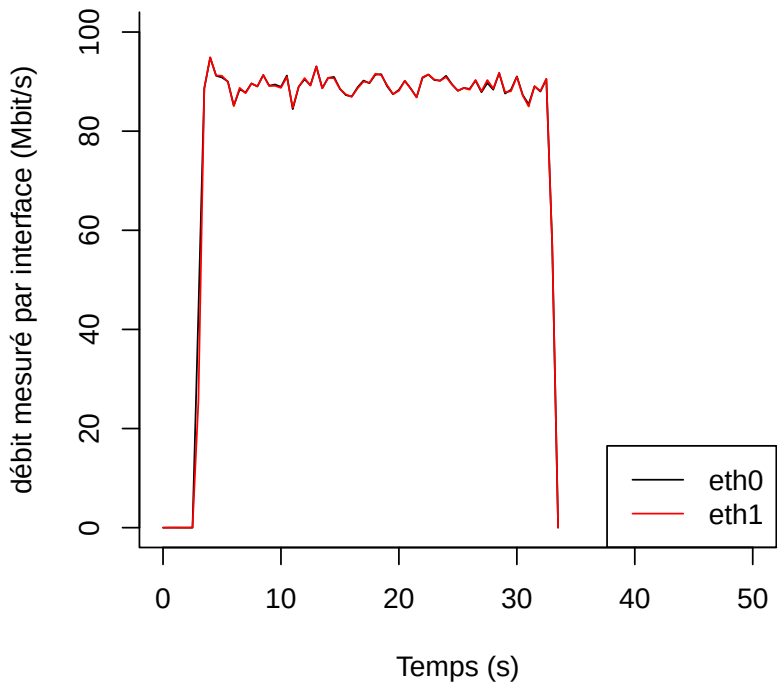

Figure 8 – **Débit entrant côté serveur.** échantillonage : 2 Hz.

Pour mesurer le débit total généré, nous moyennons les 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 obtenons, côté serveur et pour cet exemple, un débit maximal de 168 Mbit/s ce qui est attendu par les mesures de débit *via* bwm_ng.

7.4.2 Variation du débit maximal par lien

Pour connaître les limites de la capacité de notre simulation, nous faisons varier le débit maximal par sous-flot, ainsi que le nombre de sous-flots dans la topologie A.

Nous observons une phase linéaire où l'augmentation du débit par lien ou l'augmentation du nombre de sous-flots augmente linéairement le débit total obtenu côté serveur. Cette phase située en dessous des 100 Mbit/s par lien correspond au but de MPTCP : augmentation du débit. Cependant, les performances décroissent rapidement et tombent sous les performances d'une simple connexion TCP ce qui est contraire à la nature même de MPTCP [1].

Ce problème pourrait être expliquer par l'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, on obtient une taille minimale

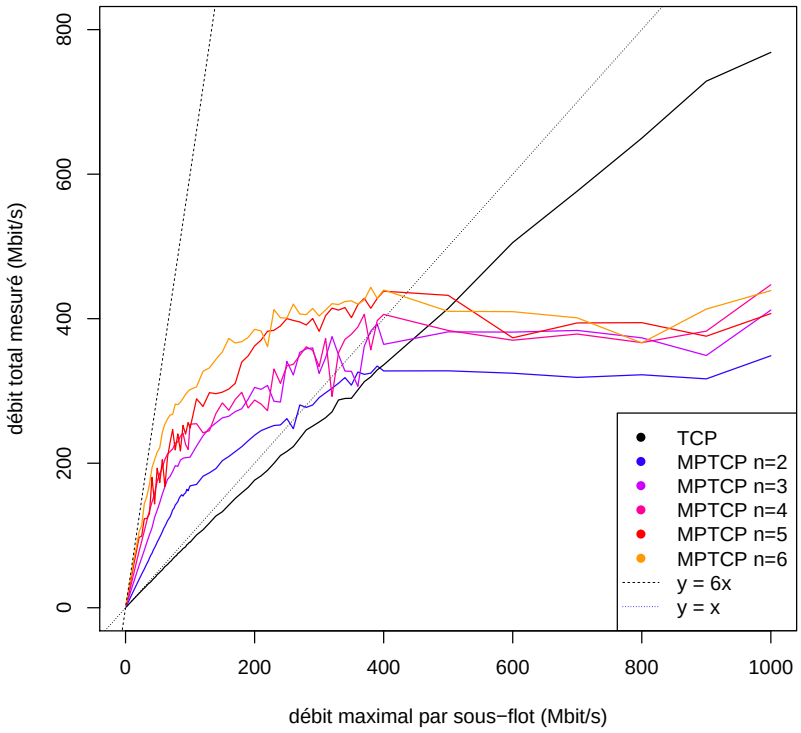

Figure 9 – **Débit total mesuré en fonction du débit maximal par sous-flot.** Les débits sont mesurés avec *iperf* pour une connexion TCP classique et une connexion MPTCP contenant de 2 à 6 sous-flots.

de tampon de 5,5 Mo. Sachant que le tampon de réception est partagé pour tous les sous-flots d'une connexion MPTCP [2], la taille minimale du tampon de réception doit suivre cette formule :

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

C'est à dire que la taille du tampon de réception doit être le produit du RTT maximal parmi tous les sous-flots et le débit total de tous les sous-flots réunis. 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 paquet, la valeur minimale correspond au BDP le 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 :

```
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 Fig. 10.

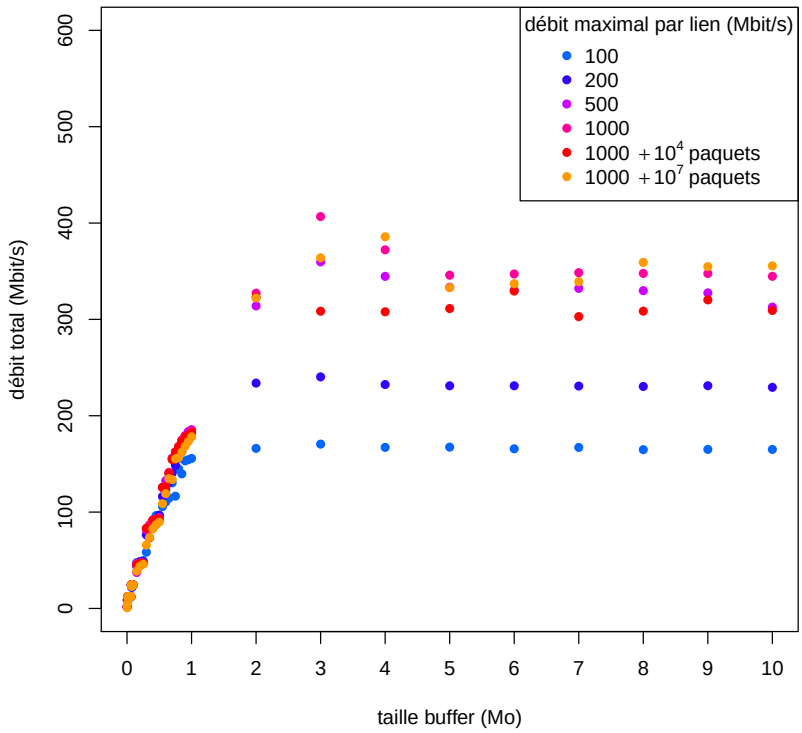

Figure 10 – **Débit mesuré en fonction de la taille de la fenêtre**. La taille maximale de la fenêtre TCP est modifiée avec l'argument '-w' avec *iperf*. La taille obtenue est étrangement de deux fois supérieure à la taille demandé. Le nombre de paquets correspond à la taille maximale de la queue des routeurs.

Nous obtenons les mêmes résultats en modifiant la taille maximale de la queue des routeurs ou en modifiant les valeurs dans le noyau (voir 8.1.3 page 26) 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 à 100 Mbit/s. C'est pourquoi, nous limiterons le débit maximal par lien à environ 10 Mbit/s.

7.4.3 Variation du délai par lien

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

Pour mesurer le délai pour chaque sous-flot, nous avons utilisé la commande *ping*. Cette approche a été utilisé 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 par TCP car l'utilisation conjointe de *ping* et d'*iperf* produit une erreur (voir 8.7.2 page 29).

Nous observons que la modification du délai n'entraîne pas de baisse de performances pour les liens à faible débits Fig. 11. Ce qui est attendu vu que la taille des *buffers* et la taille maximale de paquets attendant sur les routeurs sont largement suffisantes pour pallier à une augmentation de débit. Il reste alors à tester ces cas dans des conditions plus stringentes.

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

Nous constatons que la taille du tampon s'avère être un paramètre primordial dans pour les performances de MPTCP. L'algorithme de congestion utilisé ne semble pas être nécessaire dans ces cas particuliers. Dans une utilisation mobile, les deux sous-flots ont généralement des délais différents si on se base sur l'utilisation d'un accès wifi, 3G ou/et ethernet. Dans la figure 12, nous utilisons les mêmes paramètres que dans l'expérience précédente cependant le premier sous-flot a un délai (RTT) fixe d'une dizaine de millisecondes (expériences "TC"). Nous observons aucune différence entre le cas où les deux sous-flots ont un délai variable et le cas où un seul sous-flot est variable sauf dans l'expérience "200 TC" où le débit est plus élevé que dans le cas à deux sous-flots variables ce

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

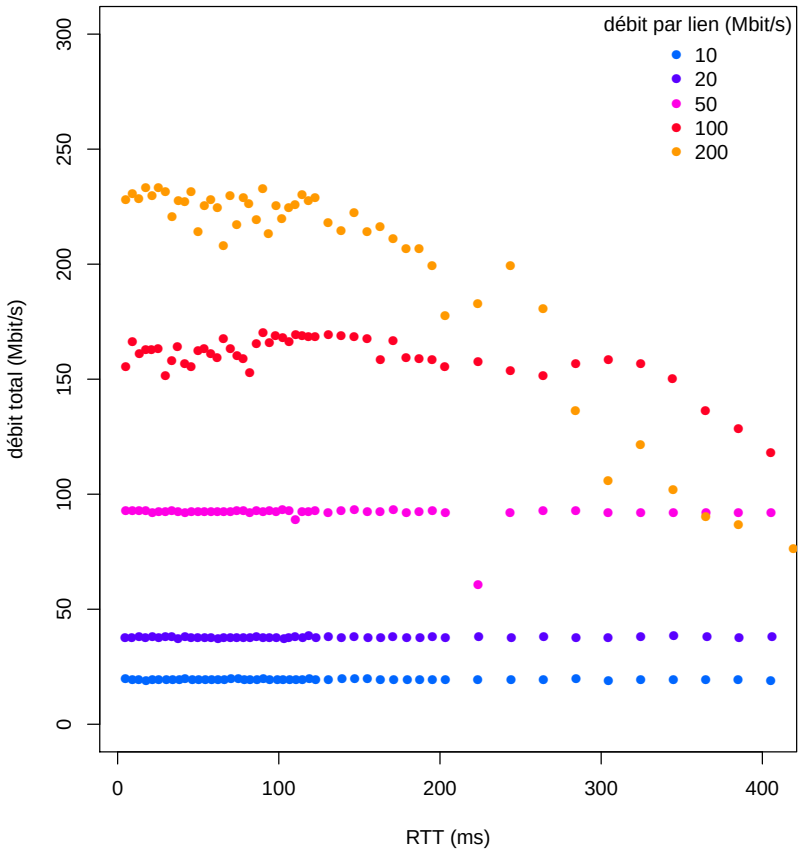

Figure 11 – **Débit maximal mesuré en fonction du délai**. Le délai est du même ordre de grandeur pour tous les liens. Le RTT est mesuré avec la commande *ping* avant le test avec *iperf* qui dure 60 s. Deux sous-flots sont utilisés.

qui confirme en partie le fait que la diminution de performance observée, pour ce cas, est lié à la taille du tampon.

7.4.4 Choix du sous-flot en fonction du délai

Pour tester les sous-flots utilisés dans une connexion à « faible » débit. Nous avons utilisé 5 sous-flots à délai variable. Le délai 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 :

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

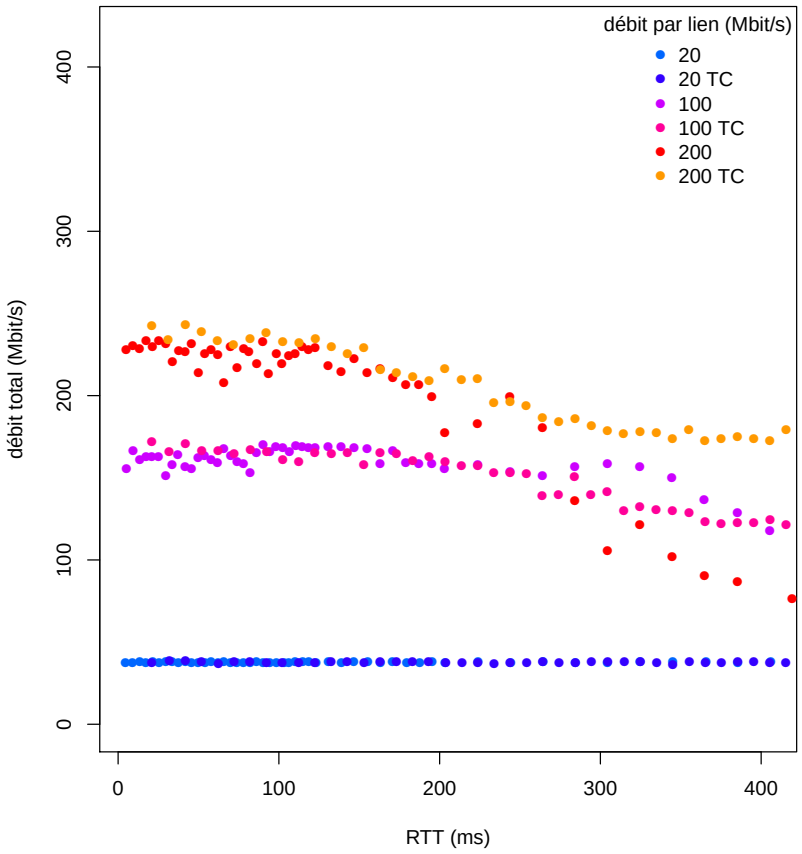

Figure 12 – **Débit maximal mesuré en fonction du délai d'un lien**. Le délai est modifiée pour le second lien dans la topologie A. Le RTT est mesuré avec la commande *ping* avant le test avec *iperf* qui dure 60 s. Deux sous-flots sont utilisés.

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

La commande *iperf* envoie des paquets jusqu'à atteindre le débit maximum sur chaque sous-flot. Il est donc difficile de mesurer quels sont les sous-flots préférés par l'ordonnanceur. *iperf* permet seulement de contraindre le débit d'UDP. Nous avons donc intégré *iperf3*[1] à la VM mininet pour permettre de fixer le débit envoyer au serveur.

des points d'inflexion.

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

L'utilisation d'*iperf3* entraîne une erreur qui n'a pas été encore résolue à la fin de la simulation (voir 8.7.2 page 29) empêchant toutes simulations automatisées. Nous avons donc choisi deux résultats caractéristiques.

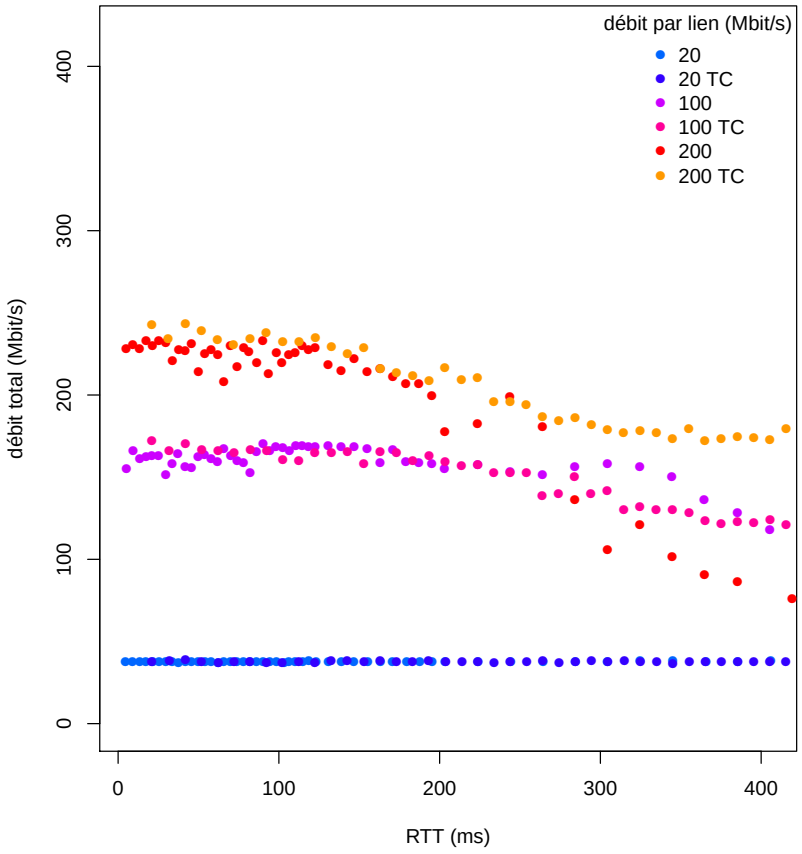

Figure 13 – **Débit maximal mesuré en fonction du délai d'un lien**. Le délai est modifiée pour le second lien dans la topologie A. Le RTT est mesuré avec la commande *ping* avant le test avec *iperf* qui dure 60 s. Deux sous-flots sont utilisés.

7.4.5 Choix de l'algorithme

La plupart des tests ont aussi été effectués avec l'algorithme de congestion de linux (*cubic*) et certains avec l'algorithme *OLIA*. L'utilisation de ces algorithmes entraînent des résultats grossièrement similaires cependant, des analyses plus poussées et des tests dans des conditions de plus grand stress devront être effectués.

8 Annexe1

8.1 Utilisation des scripts pythons

8.1.1 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".

8.1.2 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.

8.1.3 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*.

8.2 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 8.4.

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 8.4.

8.3 Résumé des arguments pour le parseur

Les commentaires des arguments sont disponibles dans la section 8.4.

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

Description

--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, -D** n	fixe le délai de tous les liens
--dump	lance tcpdump sur toutes les interfaces utilisant mptcp et écrit la sortie dans un fichier de type [hôte]-[if].pcap
–**file, -F** <filename>	fichiers de sortie, un pour le client et un pour le serveur. Par défaut, aucun fichier n'est créé.
--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
--tc	active la modification des liens via *tc*

exemple :

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

8.4 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.

8.4.1 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.

8.5 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
```

8.6 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.

8.7 Bugs

8.7.1 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.

8.7.2 mininet

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.

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

8.8 Expériences

Description des fichiers pythons dans le dossier experiment

TCPvsMPTCP/
- __init__.py
- pyMPTCP.py
- **experiment**/
  - __init__.py
  - experiment.py
  - experiment001_TC.py
  - ...

fichier	**Commentaires**
experiment.py	fixe délai h1-eth1, valeur à modifier dans le fichier
exp001_TC.py	fixe délai h1-eth1, valeur à modifier dans le fichier
exp002_TCping	même caractéristiques que exp001, le ping est mesuré pendant la mesure de débit.
exp003_TCpingws	cf exp002. arg1 est utilisé pour modifier la fenêtre TCP dans iperf « -w n »

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, 7 et 18.)

[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 19.)

[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 9.)

[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, 9, 12, 14 et 16.)

[5] R. Stewart, "Stream control transmission protocol," *RFC 4960*, September 2007. (Cité en page 7.)

[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 7.)

[7] C. Raiciu, M. Handly, and D. Wischik, "Coupled congestion control for multipath transport protocols," *RFC 6356*, October 2011. (Cité en pages 8 et 17.)

[8] 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 9.)