Rapport TP RXA

Introduction :

Les 3 indicateurs majeurs de performance réseau et comment ils interagissent pour des flux TCP et UDP sont:

Latence réseau : c'est le temps nécessaire pour véhiculer un paquet au travers d'un réseau.
La latence peut être mesurée de plusieurs façons : aller-retour (round trip), unilatérale (one way), etc…
La latence est impactée par tout élément dans la chaîne utilisée pour transporter les données : station cliente, liens WAN, routeurs, LAN / réseau local, serveur et en dernier ressort, elle est limitée pour les très grands réseaux par la vitesse de la lumière (fibre optique)
Débit réseau : il est défini par la quantité de données envoyées et reçues par unité de temps.
Perte de paquets : il s'agit du nombre de paquets perdus par rapport à 100 paquets émis par un hôtes sur le réseau.

Protocole UDP :
UDP est utilisé pour transporter des données sur des réseaux IP. Un des principes de l'UDP est qu'il fait l'hypothèse que tous les paquets émis sont effectivement reçus par l'autre partie (ou les contrôles de la bonne réception et l'éventuelle ré-émission sont gérés à un autre niveau, par exemple, par l'application elle-même).

En théorie ou pour certains protocoles spécifiques (où aucun contrôle n'est effectué à un autre niveau – par exemple pour des transmissions uni-directionnelles), la vitesse à laquelle les paquets sont envoyés par l'émetteur ne sont pas impactés par le temps nécessaire pour atteindre l'autre partie (ce qui correspond à la latence réseau). Quel que soit ce temps, l'émetteur enverra un certain nombre de paquets par seconde, qui dépend d'autres facteurs (application, système d'exploitation, ressources systèmes, …).

Protocole TCP :
TCP est un protocole plus complexe qui intègre un mécanisme qui vérifie que chaque paquet est correctement reçu. Ce mécanisme est nommé acquittement : il consiste, pour le destinataire, à envoyer un paquet spécifique ou à indiquer à l'émetteur la bonne réception de chaque paquet.

Pour des raisons d'efficacité, on n'acquitte pas chaque paquet un par un : l'émetteur n'attend pas l'acquittement du paquet précédent pour envoyer les paquets suivants. En fait, le nombre de paquets qui peuvent être envoyés avant de recevoir l'acquittement correspondant est géré par une valeur appelée « fenêtre de congestion TCP ».

Fenêtre de congestion:
Il faut comprendre que l'algorithme TCP ne connaît jamais le débit optimal à utiliser pour un lien : d'ailleurs il est difficile à estimer. IP, qui porte TCP, ne garantit pas que le chemin sera stable dans le temps à travers le réseau, d'où une impossibilité à prédire le débit à utiliser. De plus le débit est aussi conditionné par d'autres facteurs comme l'existence de flux concurrents sur une partie du chemin (par exemple vous pouvez avoir plusieurs téléchargements simultanés sur des serveurs ayant différents niveaux de performance). C'est ainsi que TCP va essayer de deviner le meilleur débit à utiliser en essayant de toujours augmenter le débit jusqu'à la survenue d'une perte de paquet parce que le réseau n'arrive pas à absorber tout le débit. En effet, les réseaux informatiques sont conçus pour être les plus simples possibles et c'est la seule possibilité qu'a un réseau pour avertir les utilisateurs qu'il est saturé.

Contrôle de flux
Le contrôle de flux, dans un réseau informatique, représente un asservissement du débit binaire de l'émetteur vers le récepteur.
Quand une machine qui a un débit montant supérieur au débit descendant de la destination, la source diminue son débit pour ne pas submerger le puits de requêtes (obligeant parfois, vu que le puits ne peut pas les traiter, la réémission de ces dernières). Le contrôle de flux doit être différencié du contrôle de congestion, qui est utilisé pour contrôler le flux de données lorsque la congestion a déjà eu lieu. Les mécanismes de contrôle de flux sont classés par le fait que le récepteur envoie une information de retour à l'émetteur.

Si nous faisons l'hypothèse qu'aucun paquet n'est perdu pour les tests : l'émetteur enverra un premier quota de paquets (correspondant à la fenêtre de congestion TCP) et quand il recevra le paquet d'acquittement, il augmentera la valeur de la fenêtre de congestion ; progressivement le nombre de paquets qui peuvent être envoyés sur une période données augmentera (débit). Le délai jusqu'à ce que le paquet d'acquittement soit reçu (correspondant à la latence) aura donc un impact sur la rapidité à laquelle la fenêtre de congestion TCP augmentera (et donc sur le rythme auquel le débit s'accroît).

Quand la latence est élevée, cela signifie que l'émetteur passe plus de temps en attente (sans envoyer de nouveaux paquets) ce qui réduit la vitesse à laquelle le débit s'accroît.

La perte de paquets aura deux effets sur la vitesse de transmission des données :

- Les paquets devront être retransmis (même si le paquet d'acquittement a été perdu et les paquets de données correctement livrés)
- La fenêtre de congestion TCP ne permettra pas un débit optimal.

Versions TCP courantes
linux :
avant 2004 : BIC avec SACK (et timestamps ?)
depuis 2004 : window scaling par défaut
depuis 2006 : CUBIC et ABC par défaut
depuis 2012 : ajout de PRR
windows :
xp (2001) : ???, pas de window scaling
vista (2007) : CTCP, window scaling par défaut
7 (2009), 8 (2012), 10 (2015) : ??
OS X : 10.10 (2014) : utilise CUBIC il paraît, window scaling, ...
Android : basé sur linux

Les critères d'évaluation des algorithmes TCP
Ces différents critères sont :
augmentation linéaire/exponentielle du débit ;
diminution linéaire/exponentielle du débit après un événement (perte de paquets, dépassement d'un seuil, …) ;
l'existence de seuils de débit dans l'algorithme (par exemple l'augmentation peut devenir prudente une fois qu'elle arrive au niveau de sa dernière meilleure performance).
Il n'est pas aisé de parler de meilleure version TCP : il y a des versions adaptées aux réseaux très hauts débits, il y a des versions adaptées aux petits débits, il y a des versions adaptées aux réseaux qui font beaucoup d'erreurs.
Enfin, on observe que les algorithmes TCP sont tous compatibles entre eux (puisqu'il n'y a pas de modification du segment TCP mais seulement une variation sur leur vitesse d'arrivée). Par exemple, Windows Vista utilise Compound TCP (en) alors que les noyaux Linux utilisent TCP CUBIC (en) depuis la version 2.6.19.

Présentation du Projet :

Le projet consiste dans un premier temps à créer un outil de benchmark réseau.
Il a été implémenté en java socket, une partie serveur qui s'occupe de répondre au requêtes du client.
On va générer deux graphiques dans un premier temps:
- Un graphique mesurant le débit en nbr octets/seconde en fonction du nombre d'octet envoyés par connection.
- Un graphique mesurant le débit en nbr octets/seconde en fonction du nombre de thread du client connectés au serveur.

A chacuns des tests, on effectuera les manipulations :

- En local
- via une connection point à point (2 terminaux relié via un Ethernet par exemple)
- via un réseau local avec switch
- via un réseau local avec hub (avec et sans collisions)
- via un réseau distant (terminaux séparé par un routeur)

Création de l'outils de benchmark

Première phase test : (optimisation de TCP)
Pour savoir quel est l'algorithme TCP utiliser sur votre machine Linux il faudra taper la commande suivante:
`cat /proc/sys/net/ipv4/tcp_congestion_control`
par défaut le résultat sera :
`cubic`

CUBIC a une phase montante douce ce qui le rend plus « amical » envers les autres versions de TCP. CUBIC est plus simple car ne possède qu'un seul algorithme pour sa phase montante, et n'utilise pas les acquittements pour augmenter la taille de la fenêtre, on préfèrera parler d'événement de congestion. C'est ce qui rend CUBIC plus efficace dans les réseaux à bas débit ou avec un RTT court.

La liste des algorithmes disponibles (inclus) :

BIC (kernel module)
New Reno (kernel module)
CUBIC (kernel module)
HSTCP (kernel module)
HTCP (kernel module)
Hybla (kernel module)
Illinois (kernel module)
Vegas (kernel module)
Veno (kernel module)

Westwood+ (kernel module)
YeAH (kernel module)

Pour changer d'algorithme , il suffit de taper la commande suivante :

```
echo nomdealgotcp > /proc/sys/net/ipv4/tcp_congestion_control
```

sur les machine de salle réseaux , il faudra configurer le noyau linux afin de réduire réduire le TIME_WAIT des sockets TCP sur le poste client avec la commande suivante:

```
cat /proc/sys/net/ipv4/tcp_fin_timeout
cat /proc/sys/net/ipv4/tcp_tw_recycle
cat /proc/sys/net/ipv4/tcp_tw_reuse
```

Vous devriez obtenir, respectivement, les valeurs 60 pour le timeout, pour le recyclage et pour la réutilisation.
Nous allons modifier ces valeurs pour réduire le timeout à 30 secondes, et recycler et réutiliser nos connexions :

```
echo 30 > /proc/sys/net/ipv4/tcp_fin_timeout
echo 1 > /proc/sys/net/ipv4/tcp_tw_recycle
echo 1 > /proc/sys/net/ipv4/tcp_tw_reuse
```

sysctl net.inet.tcp : liste des paramètre TCP

• `sysctl net.inet.tcp.rfc1323=0`, désactive l'option window scale permettant de donner un facteur du champ win (intéressant pour un réseau à forte latence et/ou haut débit).

• `sysctl net.inet.tcp.inflight.enable=0` , désactive une implémentation du contrôle de congestion de TCP proche de Vegas.

• `sysctl net.inet.tcp.newreno=0` : désactive la politique de contrôle de congestion à la NewReno.

• `sysctl net.inet.tcp.hostcache.expire=0` : Avec cette option, TCP conserve en mémoire des paramètres pour une machine destinatrice (RTT, RTTvar, fenêtre de congestion...). Au passage on peut observer ces paramètres grâce à `sysctl net.inet.tcp.hostcache.list`

• `sysctl net.inet.tcp.sack.enable=0` , permet de désactiver le mécanisme d'acquittement sélectif possible dans TCP.

• `sysctl net.inet.tcp.delayed ack=0` : cette option permet d'ajouter à TCP un délai avant émission d'un acquittement afin de minimiser le nombre d'acquittement. Nous le supprimons afin de faciliter l'analyse du contrôle de congestion à travers la capture des paquets.

• `sysctl net.inet.tcp.rfc3390=0`, désactive l'option de TCP qui ajuste la taille de la fenêtre initiale de congestion lors du slow start.

• `sysctl net.inet.tcp.sendbuf auto=0`, désactive une option permettant à TCP d'ajuster la taille du buffer d'émission lors d'une connexion.

• `sysctl net.inet.tcp.recvbuf auto=0`, désactive une option permettant à TCP d'ajuster la taille du buffer de réception lors d'une connexion.

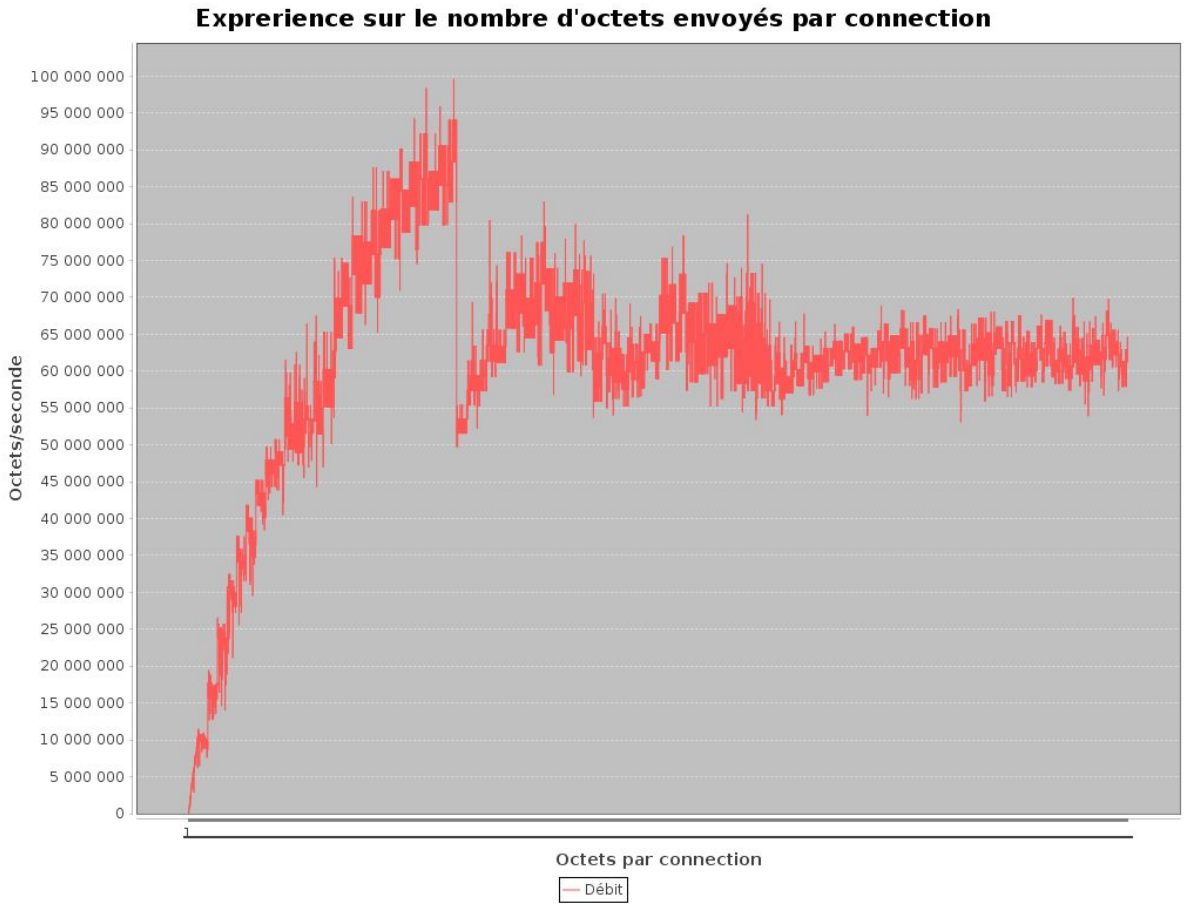
Phase d’analyse des résultats:
On a pas changer l’algorithme de congestion (par défaut cubic sur nos machines).
BIC
BIC garde un bon débit, une bonne équité même avec les autres algo d'évitement de congestion de TCP.
Une version un peu simplifiée de l'algorithme pourrait être : lorsqu'il y a une perte, BIC réduit cwnd par un certain coefficient. Avant de réduire on va garder en mémoire la valeur de cwnd qui sera notre maximum (Wmax), et la nouvelle valeur sera notre valeur minimum (Wmin). À partir de ces deux valeurs on va rechercher la valeur intermédiaire pour laquelle nous n'avons pas de pertes (recherche dichotomique).
L'algorithme fait attention à ne pas trop augmenter cwnd, aussi si lors de notre recherche on fait des sauts trop grands (défini par un certain seuil), on préfèrera accroître cwnd de manière logarithmique. Une fois la différence amoindrie, on pourra repasser en recherche dichotomique jusqu'à arriver à Wmax. À partir de là on recherche un nouveau maximum, avec une croissance linéaire : on cherche s'il y a un maximum proche. Si après une certaine période on n'a pas de perte, alors on se dit que le maximum est bien plus loin et on augmente de nouveau la taille de la fenêtre de manière exponentielle.
BIC permet une bonne équité, il est stable en maintenant un haut débit et qu'il permet une bonne mise à l'échelle. Malgré ses qualités, il reste trop agressif durant sa phase montante. Mais surtout il provoque une inégalité avec les flux ayant des RTT longs.
Cet algorithme a été utilisé dans Linux, depuis on lui préfère sa variante plus élaborée: CUBIC.

CUBIC a une phase montante douce ,ce qui le rend plus « amical » envers les autres versions de TCP. CUBIC est simple car ne possède qu'un seul algorithme pour sa phase montante, et n'utilise pas les acquittements pour augmenter la taille de la fenêtre, on préfèrera parler d'événement de congestion. C'est ce qui rend CUBIC plus efficace dans les réseaux à bas débit ou avec un RTT court.

Compound TCP
Une des grandes particularités de CTCP est qu'il maintient deux fenêtres de congestion. La première est la fenêtre qui augmente de façon linéaire mais décroît en cas de perte via un certain coefficient. La seconde fenêtre est liée au temps de réponse du destinataire. On combine donc des méthodes liées à la perte de paquet (comme TCP Reno) et d'adaptation préventive à la congestion (comme TCP Vegas), d'où le nom « Compound » (composé).
La taille de la fenêtre réellement utilisée est la somme de ces deux fenêtres. Si le RTT est bas, alors la fenêtre basée sur le délai augmentera rapidement. Si une perte de segments survient, alors la fenêtre basée sur la perte de paquet diminuera rapidement afin de compenser l'augmentation de la fenêtre basée sur le délai. Avec ce système, on cherchera à garder une valeur constante de la fenêtre d'émission effective, proche de celle estimée.
Le but de ce protocole est de maintenir de bonnes performances sur des réseaux avec un haut débit et avec un grand RTT.

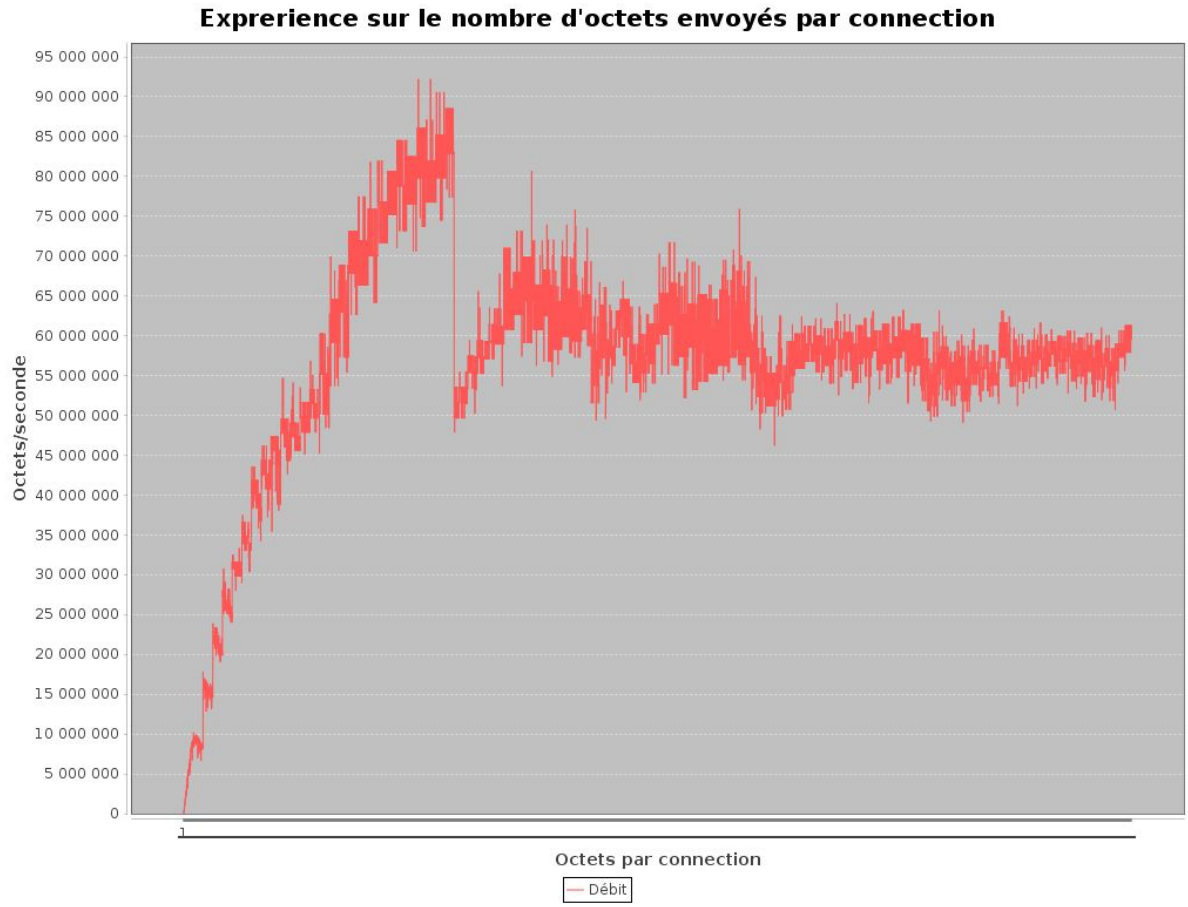
Test une seule connexion:

On envoie 1 Ko de données pour chaque test jusqu'à atteindre 1 Ko , on change la valeur à 1Mo et on continue les tests.
On commence par envoyer 1 octet par connexion et on lance 10 fois le test.
Puis on passe à 2 octets par connexion et on lance 10 fois le test
...
On s'arrête quand on envoie 1Ko par connexion et lance 10 fois.



Test PC=>PC avec un lien de 1Gb/s

On remarque le débit augmente en douce (simulation de slow start tcp avec notre algorithme).
Lors de la première phase, on remarque une sorte d'escalier et de baisse de débit, c'est lié à la taille du buffer que j'ai choisi.



Test PC=>SWITCH=>PC avec un lien de 1Gb/s

On remarque que les deux courbes sont presque identiques , car la vitesse du lien sur toutes les parties est de 1Gb/s.

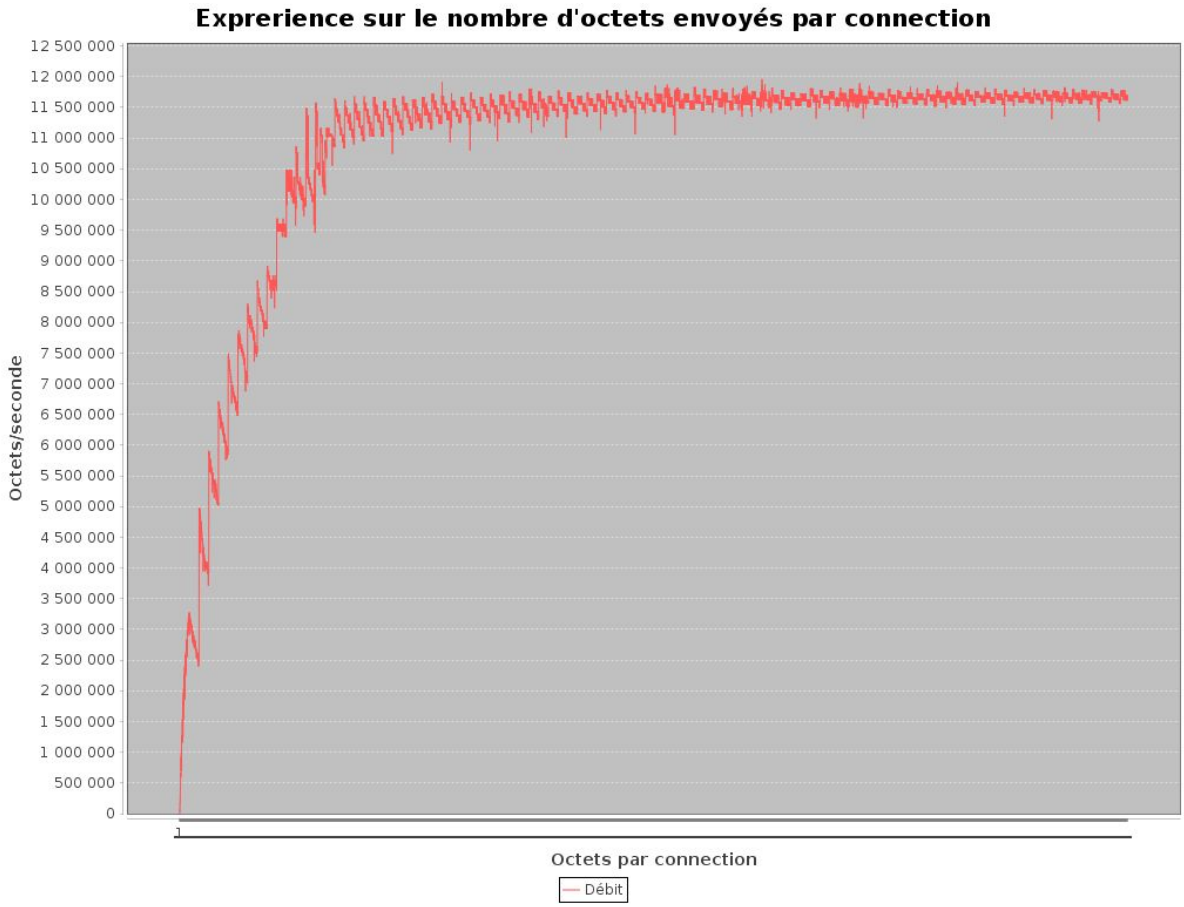
Pour résoudre le problème de l'escalier , nous avons effectuer des analyse plus précises en zoomant sur les parties concernées, nous nous somme rendu compte que c'était la taille du buffer qu'on a fixer qui posait problème.on décider de laisser Java s'occuper de la taille automatiquement.

PC (WIN) => PC(LINUX)

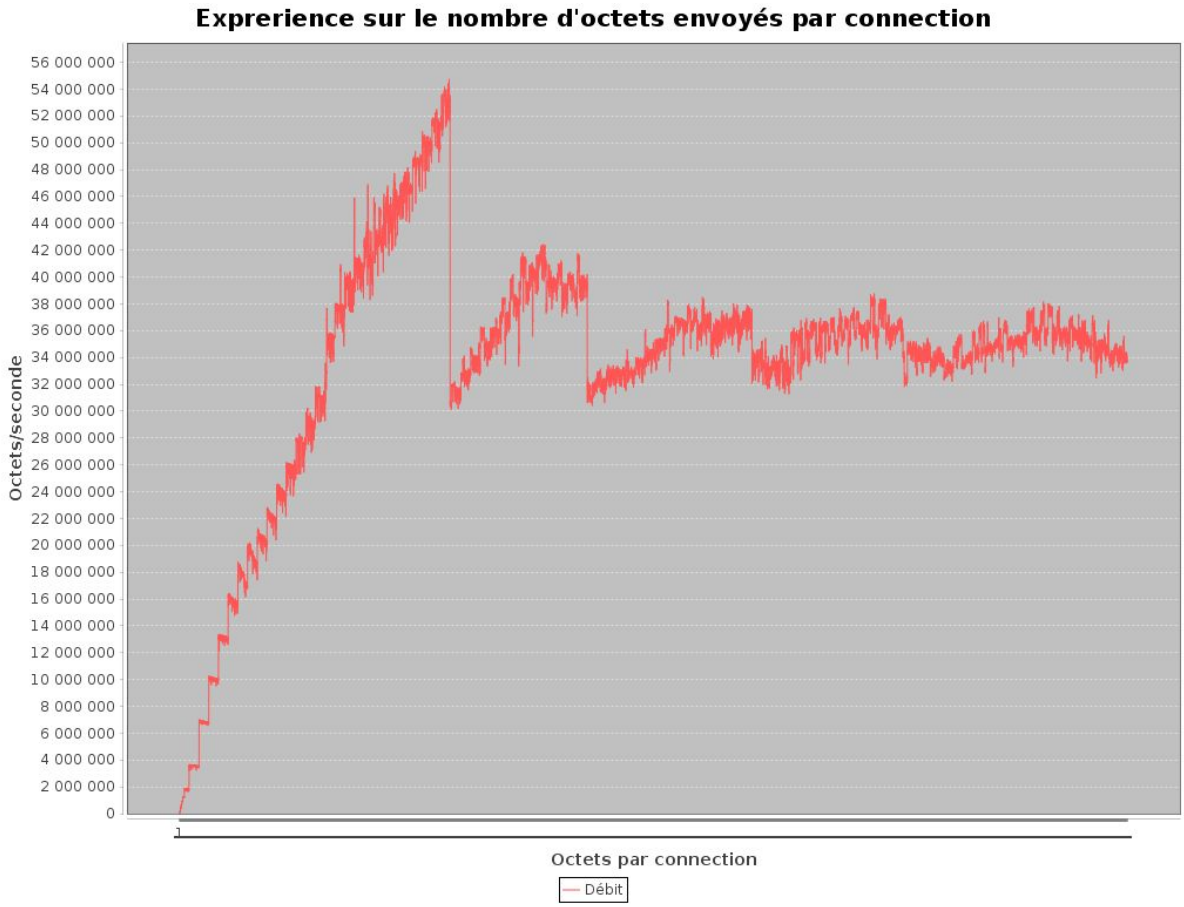
On remarque une net amélioration de la courbe et la montée presque linéaire est dû à l'utilisation de Windows , sur Linux la montée est logarithmique.

Le problème de chute de débit persiste toujours, on a fait beaucoups de recherche pour expliquer ce phénomène:

Lorsque on atteint la taille de fenêtre tcp maximale, on envoi un octet de plus et on coupe la connexion, le serveur n'a pas le temps de répondre et d'ajuster la taille de sa fenêtre. C'est ce qui cause une grosse chute débit.



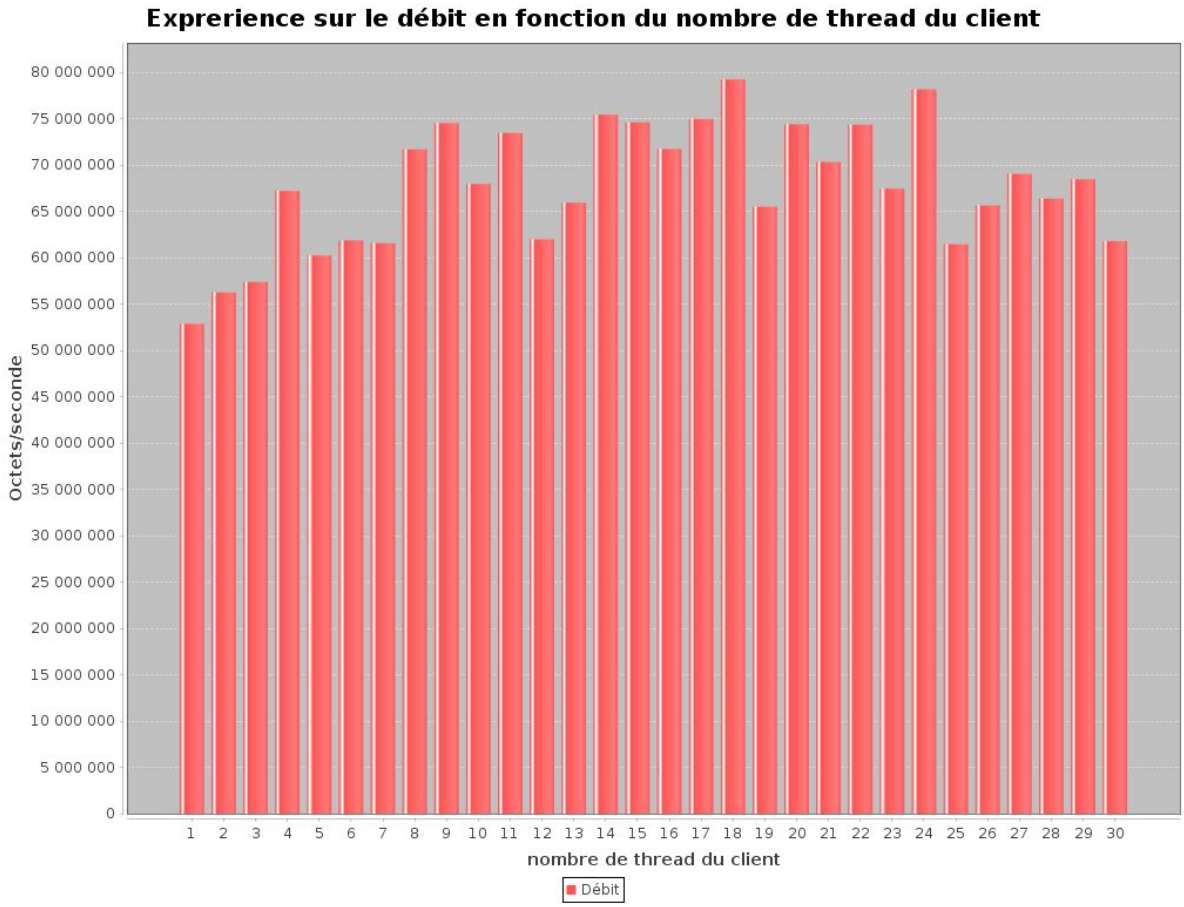
Test PC=>SWITCH=>PC avec un lien de 100Mb/s



Test PC=>ROUTER=>PC avec un lien de 100Mb/s

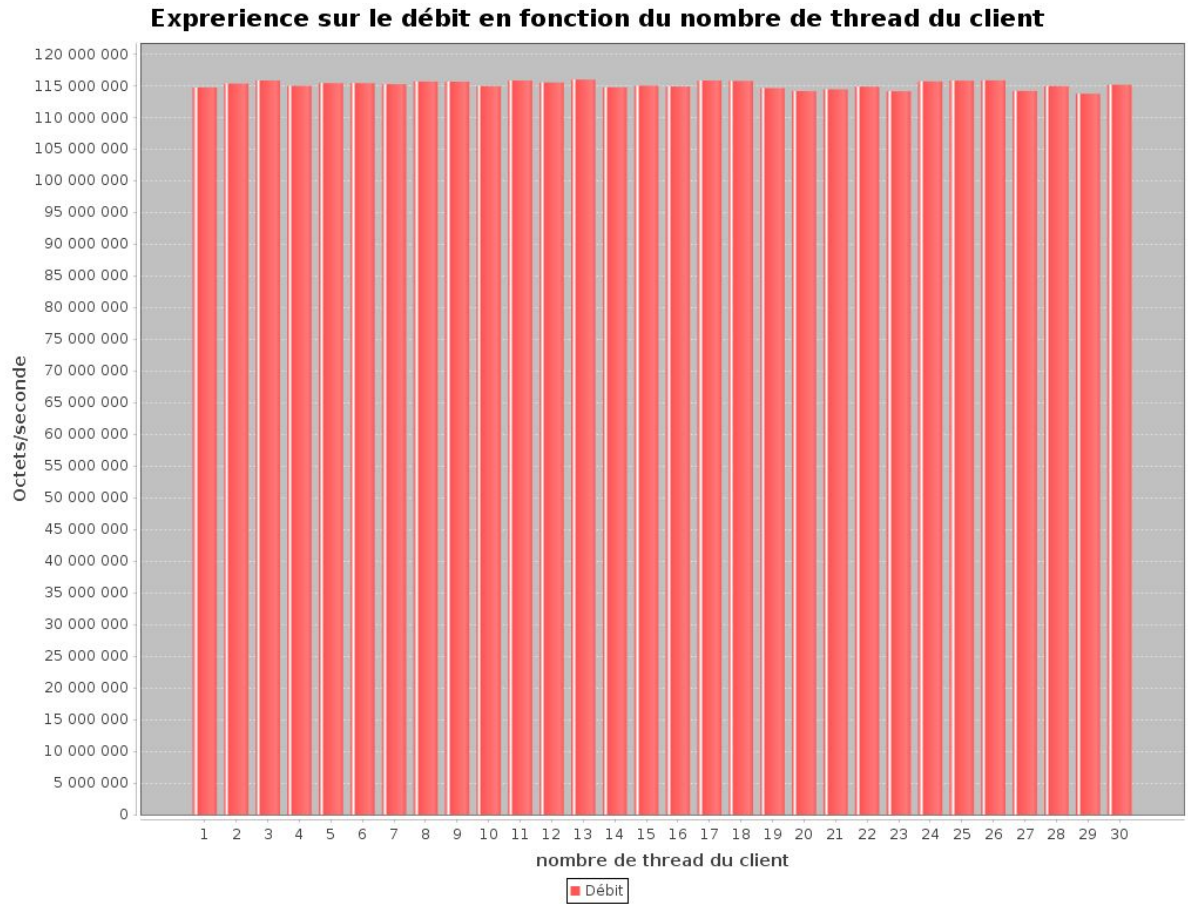
Test Thread:

On envoie 1Go de données au serveur en utilisant 1 thread et on répète 10 fois puis la même chose avec 2 threads … (max de threads 30).
Le 1Go de données sont toujours découper par rapport au nombre de threads ex:
1Go/4 threads => chaque thread va envoyer 250Mo et va fermer la connexion.
Hypothèse:
Cette stratégie est efficace en terme d'utilisation de la bande passante, car elle utilise toujours le maximum, mais le problème est qu'elle surcharge le serveur, et quand les fichiers sont très volumineux, l'assemblage prend beaucoups de temps .



Test PC=>LOCALHOST avec un lien de 100Mb/s

Les thread utilise la vitesse maximale de transmission.du lien.



Test PC=>PC avec un lien de 1Gb/s

remarques :

1. le test en pair-to-pair donne un résultat avec la vitesse maximale.
2. les test avec switch avec une liaison à 1Gb/s donnent un résultat linéaire avec une perte de débit.
3. les test avec switch avec une liaison à 100 Mb/s donnent un résultat logarithmique car la montée jusqu'à la vitesse maximale prend moin de temps que pour arriver à 1Gb/s.

La vitesse avec le mode thread reste sensiblement la même sauf sur un HUB, au début avec 1 seul et 2 threads , le débit est bas , à partir de 3 threads la vitesse maximale est atteinte.

Conclusion:
Après la fin des deux tests en mode connexion et threads, on constate que le mode multi-thread est plus performant pour le transfert de données en mode TCP.

Sources :

https://fr.wikipedia.org/wiki/Algorithme_TCP
http://blog.performancevision.com/fr/performance-r%C3%A9seau-liens-entre-latence-d%C3%A9bit-et-perte-de-paquets
http://eugen.dedu.free.fr/teaching/tcp/cours.pdf
http://sgros.blogspot.fr/2012/12/controlling-which-congestion-control.html
https://www.skyminds.net/serveur-dedie-reduire-les-connexions-time_wait-des-sockets-et-optimiser-tcp/
http://lig-membres.imag.fr/sicard/tpRES/tcpRICM2congestion.pdf
http://www.cisco.com/c/en/us/td/docs/switches/datacenter/mds9000/sw/nx-os/configuration/guides/qos/qos_cli_4_2/qos_cli_4_2_cg/qosov.html#wp75144
https://dpt-info.u-strasbg.fr/~mathis/Enseignement/Reseau/TP_NS/ns_tcp.html
http://c3lab.poliba.it/TCP_over_Hsdpa
http://www.linux-admins.net/2010/09/linux-tcp-tuning.html