Comparaison des principaux algorithmes TCP

Thibaut Ehlinger, *Université de Strasbourg*

20 octobre 2015

TCP (*Transmission Control Protocol*) est une implémentation de la couche transport du modèle OSI. C'est de loin son implémentation la plus populaire, avec pour maigre concurrent UDP, qui est moins contraignant et donc plus utile pour les connexions ne nécéssitant pas une grand robustesse. TCP étant omniprésent dans les réseaux actuels, il a donc souvent été étudié et réimplémenté, parfois parce qu'une implémentation présentait des faiblesses qui lui étaient inhérentes, parfois parce qu'aucune implémentation ne répondait à une problématique bien précise du réseau l'utilisant, comme c'est le cas pour les *long fat network*, par exemple. Dans cet rédaction, nous étudierons tout d'abord les différentes implémentations historiques de TCP. Celles ayant été utilisées un jour ou l'autre mais qui sont ajourd'hui obsolètes car peu efficaces. Puis nous étudierons certains des algorithmes utilisés aujourd'hui, dans le but de comprendre quels sont les critères déterminants lors du choix de l'algorithme utilisé. En effet ceux-ci peuvent-être plus ou moins "aggressifs" ou stables.

Les premières implémentations de TCP

De TCP Tahoe à TCP Westwood, en passant par TCP Reno, nous allons vous présenter une série d'algorithmes qui se sont globalement succédés, chacun étant dans la plupart des cas plus efficace que son prédecesseur, en tout point.

Tahoe

Slow start & congestion avoidance

Cette implémentation est citée pour la première fois en 1988 dans un article de Van Jacobson et Karels. C'est la plus simple et la moins efficace, tout types de problème confondus. Au début, on est en **Slow Start** : tant qu'on n'a aucune perte, on double la taille de la *congestion window* (**CWND**), c'est à dire la taille en segments des rafales TCP envoyées. À chaque fois que tous les segments envoyés sont acquittés, on double la taille de la CWND. Le slow start est interrompu dès la première perte de paquet, c'est à dire lorsqu'on reçoit plusieurs fois le même acquittement (**ACK**) ou lors d'un **time out**.

Comme la plupart des pertes de paquets sur les réseaux filaires sont provoqués par des congestions, Tahoe implémente le mode **Congestion Avoidance (CA)**. On se base sur une variable nommée **sstresh** (*slow-start threshold*). Lorsque $CWND <= sstresh$, on est en Slow Start, sinon on est en CA. En CA, on augmente la taille de la fenêtre de 1 segment à chaque fois que tous les segments d'une rafale sont acquittés. C'est donc une progression linéaire.

Fast retransmit

Si un ACK est reçu trois fois de suite, alors le segment (ACK+1) a probablement été perdu. Il ne s'agit probablement pas d'un déséquencement. Pour pallier ce problème, Tahoe renvoie directement le segment concerné au lieu d'attendre la fin d'un timeout. Après un fast retransmit, $sstresh = \frac{CWND}{2}$ et $CWND = 1$ (Slow Start).

Avantages et faiblesses

En résumé, Tahoe est déjà une réelle avancée, car il permet d'approximer très grossièrement la plus grosse taille de fenêtre d'émission possible, et il reste stable à moins qu'il n'y aie des congestions sur le réseau. Dans ce cas, Tahoe peut s'adapter au problème et diminuer la taille de sa fenêtre d'émission.

Néanmoins, la réinitialisation de la fenêtre d'émission à **1** à la moindre perte de paquet est une sous-estimation beaucoup trop importante. La taille moyenne de CWND est donc bien plus faible que ce qu'elle pourrait être optimalement.

Reno

Reno résoud déjà grandement le problème de l'oscillation perpetuelle Slow Start/CA en ajoutant la notion de **Fast Recovery**.

Fast recovery

Après un fast retransmit, au lieu de passer en Slow Start on applique : $sstresh = \frac{CWND}{2}, CWND = sstresh + 3$. En fait on repasse directement en CA, tout en augmentant la taille de CWND de 3, en référence aux trois segments qui n'ont pas été reçus à cause des ACK dupliqués. En cas de time out néanmoins, on repasse en Slow Start. Cette stratégie permet en cas de perte de ne pas baisser drastiquement la taille de CWND, à moins d'un time out. Un time out est en fait plus grave qu'une simple perte de paquet. Il témoigne probablement d'une modification topologique du réseau ou bien d'une congestion importante, alors qu'une simple perte de paquet peut-être due à des évènements plus ponctuels, tels qu'un déséquencement ou une perte.

Faiblesses

TCP Reno réagit bien au pertes de paquets quand elles se limitent à une par rafale. Quand par contre il y en a plusieurs par rafale, Reno est presque aussi inefficace que Tahoe, car il ne peut détecter qu'une seule perte de paquet à la fois.

De plus, il se peut qu'au sein d'une même rafale, l'émetteur aie le temps de passer en Fast Recovery, puis à nouveau en Congestion Avoidance, **deux fois**. Ce qui veut dire que si la fenêtre est trop grande et les pertes trop espacées au sein de cette fenêtre, la CWND peut-être divisée par 4.

New Reno

New Reno peut détecter les pertes de paquets multiples et est, de fait, plus performant que Reno. En effet, New Reno garde une trace de tous les segments envoyés dans une rafale. Lorsqu'il détecte une perte, il **reste** en fast recovery tant que tous les segments de cette rafale n'ont pas été acquittés par le destinataire. New Reno est mieux que Reno en tout point.

Problèmes

Puisque les pertes ne sont détectéee s que par les duplications d'acquittements, il faut tout de même attendre un timeout entier pour détecter toutes les pertes.

SACK

ACK sélectifs

SACK permet de nommer les segments non-reçus, contrairement à Reno qui se contente de dupliquer les acquittement. Cela permet de récupérer plus vite les paquets perdus.

Variable *pipe*

En Fast Recovery, SACK initialise une variable *pipe*. C'est une estimation du nombre de paquets qui sont encore dans le réseau. Si SACK reçoit une duplication d'ACK, il incrémente pipe. Il le décrémente pour toute nouvelle transmission ou nouvel ACK. Quand $pipe < CWD$, il renvoie tous les segements qui n'ont pas été acquittés. Cela permet de renvoyer plusieurs segments perdus en moins d'un RTT.

Problèmes de SACK

SACK est nettement plus efficace que New Reno, mais il présente un défaut majeur : il doit être implémenté par l'émetteur **et** le récepteur, sinon il ne marchera pas.

Vegas

Mécanisme de retransmission

D'après les travaux de Lawrence Brakmo et Larry L. Peterson, deux chercheurs de l'université d'Arizona et concepteurs de Vegas, Reno et New Reno sont très peu performants en ce qui concernt les **time out**. En effet, ils ont mesuré que pour un RTT moyen de 500ms, il fallait approximativement **1100ms** à New Reno pour détecter un time out.[1] Ce qui correspond effectivement à plus de deux RTT. Vegas mesure le temps écoulé entre l'envoi d'un segment et la réception de son acquittement. Et mesure ainsi dynamiquement les RTT. Cela permet non seulement d'estimer le time out plus précisément et de perdre moins de temps en cas de réinitialisation déclenchée par un time out, mais non seulement de détecter les pertes et les déséquencements préventivement :

En cas d'ACK dupliqué , on compare le nouveau RTT avec le RTT mesuré précédemment. Si elle est supérieure à Time Out value, on renvoie directement le segment suivant, sans attendre trois acquittements dupliqués.

[1] *In Reno, round trip time (RTT) and variance estimates are computed using a coarse-grained timer (around 500 ms), [...] we found that for losses that resulted in a timeout—usually due to two or more dropped segments in a RTT , the exact figure depends on the number of segments in 4 transit—it took Reno an average of 1100ms.*

Après un ACK dupliqué on fait de même. En effet, il est peu couteux de renvoyer un segment que de perdre beaucoup de temps pour découvrir qu'il a été perdu.

De plus, Vegas ne diminue la taille de sa CWND que si le dernier ACK renvoyé a été renvoyé **après** le dernier changement de taille de fenêtre. En effet on peut supposer que deux pertes peuvent avoir été provoquées par la même congestion et qu'il ne sert à rien d'y réagir deux fois.

Congestion avoidance

Pour l'évitement de congestions, Vegas fait une estimation de la CWND *attendue* et la compare avec la CWND *estimée*. En suite on fixe de seuils $\alpha < \beta$. Si la différence des seuils est inférieure à α alors on augmente CWND linéairement, si elle est supérieure à β, on la diminue linéairement.

Congestion avoidance

En Slow Start, Vegas n'augmente exponentiellement que tout les RTT, donc moins fréquemment que ses comparses. De plus, on fixe expérimentalement un seuil γ. Si la différence entre le RTT *estimé* et le *attendu* est inférieure à ce γ, on repasse en CA.

Aggressivité

La courbe de Vegas est donc très ronde puisque l'algorithme tente d'anticiper les problèmes de congestion. Ces variations sont beaucoup plus lentes que Reno, par exemple. On dit que Vegas est **peu aggressif**. Contrairement à Reno ou à Tahoe, il ne tente pas de prendre le plus de bande passante que possible. Il en prend plus si il sent que le réseau lui permet. Si Reno intervient sur le même réseau que Vegas, alors les performances de Vegas vont diminuer considérablement, étant donné que Vegas diminuera de sa propre initiative son débit d'émission en amont de la congestion provoquée par Reno. Vegas cède en quelque sorte la priorité aux autres algorithmes cités précedemment.

Westwood

Westwood a été conçu pour les réseaux connaissant un taux de perte de paquet important malgré une absence de congestion, tels que les réseaux sans fil. Comme ces pertes sont inhérentes au réseau, il ne sert à rien de compter directement les paquets perdus, car le même taux de paquet sera perdu quelque soit la taille de CWND. Du coup, Westwood fait une estimation du débit du réseau en divisant la taille d'un paquet envoyé par l'intervalle de temps entre les deux derniers acquittements. En cas de duplication d'ACK ou de time out, on se base sur cette estimation pour fixer sstresh : $sstresh = \frac{BWE \times RTT_{min}}{segsize}$. Westwood se base donc sur le débit moyen du réseau au lieu de compter les pertes.

Aggessivité

Westwood est "amical" (*friendly*). D'après les fondateurs de Westwood, lorsqu'il est en concurrence avec Reno, il partage la connexion très équitablement. De plus, TCP Westwood est considérablement plus performant que Reno lorsqu'il y a des pertes de 1% dans le réseau, ce qui est un taux crédible dans des réseaux non-filaires.

Les implémentations modernes de TCP

BIC (*Binary Increase Congestion Control*

BIC est une implémentation TCP optimisée pour les *Long fat network*, c'est à dire les réseaux à grande latence et à à grand débit, tels que les réseaux satellitaires. Dans ces réseaux, il faut trouver la CWND optimale en provoquant le moins de perte de paquet possible. Pour ce faire, BIC fait une recherche binaire, recherche réputée pour sa complexitée logarithmique.

Protocole

Pour procéder à sa recherche binaire, BIC enregistre W_{max}, la taille maximum de CWND avant le dernire Fast Recovery, et W_{min}, la taille de la fenêtre juste **avant** le dernier fast recovery. La moyenne de W_{max} et W_{min}, W_{temp} est utilisée comme CWND. On augmente CWND linéairement jusqu'à W_{max}. S'il y a perte de paquet, W_{temp} devient le dernier W_{max}, sinon il devient le dernier W_{min}. La taille de la fenêtre $\{W_{max}\ W_{min}\}$ est divisée par deux à chaque itération. Afin de ne pas imposer de changement de débit trop violent au réseau, on plafonne l'augmentation de la CWND arbitrairement avec un indice S_{max}.

Quand CWND a atteind la granularité minimale, c'est à dire quand la taille est inférieure à S_{max} , on repasse en slow start, en TCP "normal", pour essayer de trouver une nouvelle fenêtre, qui sera peut-être plus grande.

CUBIC

Compound