CORE
Metadata, citation and similar papers at core.ac.uk
Provided by INRIA a CCSD electronic archive server

Fonctionnement de TCP: problèmes et améliorations

Omar Ait-Hellal, Eitan Altman

To cite this version:

Omar Ait-Hellal, Eitan Altman. Fonctionnement de TCP: problèmes et améliorations. RR-3603, INRIA. 1999. inria-00073076

HAL Id: inria-00073076
https://hal.inria.fr/inria-00073076

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Evolution of TCP : problems and enhancements

Omar Ait-Hellal, Eitan Altman

N° 3603

January 1999

THÈME 1

ISSN 0249-6399

Evolution of TCP : problems and enhancements

Omar Ait-Hellal, Eitan Altman*

Thème 1 — Réseaux et systèmes
Projet Mistral

Rapport de recherche n° 3603 — January 1999 — 35 pages

Abstract: In the present paper, a detailed description of some algorithms (versions) for the flow and congestion control in TCP is given. We present the versions in the chronological order, and we show the similarities between them. By using simple simulation examples, we illustrate the insufficiencies of each version, the problems outlined in some publications [13, 14] as well as those we have observed [3]. We propose enhancements for TCP New-Reno, in order to avoid some problems observed in the latter. We propose also a method which allows us to detect a loss of a retransmission, by using duplicate acknowledgments. Finally, we discuss some new mechanisms introduced by TCP Vegas, and propose enhancements (slow-start).

Key-words: TCP/IP, Internet, simulation

* INRIA sophia Antipolis, BP 93, 2004 route des luciolles, 06902 Sophia Antipolis Cedex, France. E-mail : {oaithel, altman}@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Fonctionnement de TCP : problèmes et améliorations

Résumé : Dans le présent article, une description détaillée de plusieurs versions d'algorithmes pour le contrôle de flux et de congestion dans TCP est donnée, dans l'ordre chronologique (et "logique") de leur apparition. Nous illustrons par des exemples simples de simulations les insuffisances de chaque version et les problèmes relevés dans quelques publications [13, 14], ainsi que ceux que nous avons constatés [3]. Nous proposons des solutions aux problèmes identifiés dans l'algorithme New-Reno, qui portent essentiellement sur la retransmission inutile de paquets. Nous proposons également une méthode permettant de détecter la perte d'une retransmission en utilisant les acquittements dupliqués. Nous analysons enfin quelques nouveaux mécanismes introduits par TCP Vegas, et en proposons des améliorations (slow-start).

Mots-clés : TCP/IP, Internet, simulation

1 Introduction

Les mécanismes de contrôle de congestion pour les protocoles de TCP/IP ont été longtemps étudiés et de nombreux algorithmes ont été proposés dans la littérature. Beaucoup de ces algorithmes resteront sur papier sans qu'il aient été implémentés (ex. Tri-S [31]). D'autres susciteront beaucoup plus d'intérêt, comme celui proposé par Van Jacobson [15] baptisé Tahoe. Cet algorithme, comme beaucoup d'autres, est basé sur un système de fenêtrage dynamique. Révisé deux ans après par son auteur [17], en introduisant les mécanismes de *Fast Recovery* et *Fast Retransmit* [13, 29], cet algorithme a donné naissance à une nouvelle version appelée *Reno*.

A l'issue d'études sur les insuffisances et les dysfonctionnements de Reno, une autre version baptisée New-Reno a été proposée par J. Hoe [14]. Enfin, l'une des versions de TCP qui a suscité beaucoup d'attention est TCP-Vegas, qui a été proposée dans le courant de 1994, par L. Brakmo et al [7]. Cette dernière se veut plus efficace et plus performante.

Dans le présent article, nous présentons avec plus de détails les versions citées ci-dessus, qui nous semblent retenir le plus d'intérêt de la part de la communauté réseaux et protocoles. Nous présentons également quelques problèmes connus des différentes versions, ainsi que des problèmes nouveaux que nous avons identifiés. Nous illustrons ces prblèmes par de simples exemples de somulation et nous proposons des solutions pour y palier, ainsi que des améliorations pour le contrôle de flux dans TCP.

Les simulations présentées tout au long de ce travail ont été obtenues à l'aide du simulateur de réseaux REAL [21]. Ce simulateur permet de simuler des réseaux à commutation de paquets. Nous avons inclus la version Vegas de TCP, par modification du code source jk_ftp.c (code source pour Tahoe dans REAL), et les modifications apportées s'inspirent du code de Vegas [6] (vtcp_input.c, vtcp_timer.c,...). Pour illustrer les différents problèmes et caractéristiques de chaque algorithme, nous avons considéré un modèle simple avec une source TCP contrôlée transmettant des paquets vers une destination et partageant un goulot d'étranglement avec une autre source (source Poissonnienne).

Nous avons permis une fenêtre maximale de 128 paquets, correspondant approximativement à 64 Koctets qui n'est jamais atteinte dans nos exemples. Tous les paquets ont une taille fixe de 576 octets (les entêtes IP et TCP comprises), ce qui donne une taille de fenêtre multiple de 576 octets.

L'article est structuré comme suit : Nous présentons d'abord la version Tahoe de TCP et illustrons ses insuffisances, et ce qui a donné naissance à la version Reno. Nous présentons cette dernière dans la section 3 et nous la comparons à l'ancienne. Dans la section 4, la version Vegas de TCP est décrite, et les nouveaux mécanismes introduits sont discutés. Quelques problèmes sont présentés dans la section 5, et la toute recente version de TCP (TCP New-Reno [14]) venant pour palier à ces problèmes est décrite. Nous analysons cette dernière version et proposons des améliorations. Enfin, Nous proposons une méthode permettant de recouvrir des pertes des retransmissions.

2 TCP Tahoe

Dans les années 80, des protocoles basés sur le système de fenêtrage ont été proposés. Ainsi, en utilisant des fenêtres on peut raisonnablement limiter le débit d'une source. La fenêtre fixe a été la première proposée, cependant il est évident de constater qu'avec un tel système de fenêtrage, une fois que la congestion a lieu elle demeurera pour de très longues périodes. Pour éviter cela, un mécanisme de fenêtre variable lui est préférable [15, 25, 31] pour répondre aux situations de congestion dans les réseaux (longue distance).

Ainsi apparut la première version de TCP, baptisée TCP-Tahoe. Dans cette version, la fenêtre de congestion ($Cwnd$) définit le nombre de paquets (ou octets) transmis et non encore acquittés. Le contrôle de flux et de congestion est régulé en fonction des pertes constatées par une source. Ces pertes, sont détectées par un temporisateur. En effet, si le temporisateur expire ne voyant aucun acquittement arriver durant cette période, le premier paquet (en séquence) non encore acquitté est alors considéré perdu.

La source en fait exécute deux phases de transmission : le *slow start* et la phase de *congestion* :

Phase du *Slow Start*

C'est une ouverture (croissance) "exponentielle" de la fenêtre. La source augmente la taille de sa fenêtre d'un paquet pour chaque acquittement reçu. La phase du *slow start* est initiée chaque fois qu'une perte est détectée, et elle se termine

quand la taille de la fenêtre atteint un certain seuil appelé *seuil du slow start* W_{th} (initialisé dans nos simulations à 32 paquets).

Phase Congestion

Cette phase commence à la fin de la phase du *slow start*. TCP continue d'augmenter la taille de sa fenêtre de $1/Cwnd$ chaque fois qu'il reçoit un acquittement, jusqu'à ce que la fenêtre atteigne la taille de la fenêtre de réception (Rcv_wnd) ou la taille maximale. Comme la valeur de la taille de la fenêtre que nous avons considérée dans nos simulations est entière, et que $1/Cwnd$ ne l'est pas, alors augmenter la fenêtre de $1/Cwnd$ pour chaque acquittement, revient à augmenter la fenêtre d'un paquet, chaque fois que tous les paquets de la précédente fenêtre sont acquittés (tous les RTTs). La phase de congestion prend fin dès la détection de perte (expiration du temporisateur). Dans ce cas, le seuil du *slow start* est mis a la moitié de la taille actuelle de la fenêtre et la fenêtre est remise à un, et un nouveau cycle commençant par un *slow start* est entamé.

2.1 Vers TCP Reno

Le problème majeur de Tahoe est le temps dépensé dans la détection de pertes. En effet, les temporisateurs sont généralement et même plus souvent d'une résolution de 500 ms, pendant que les délais d'aller retour tendent de plus en plus à se raccourcir (câbles torsadé, fibre optique,...). La détection d'une perte peut alors prendre au moins 500 ms ce qui constitue plusieurs délais d'aller retour.

Des améliorations ont été introduites pour permettre d'utiliser beaucoup plus la bande passante. Le mécanisme de *Fast Retransmit* en est un exemple. Ce mécanisme, considère qu'un paquet est perdu dès la réception de trois acquittements dupliqués par la source, ce qui permet de détecter les pertes plus rapidement. Le nombre de trois (comme expliqué précédemment) a été justifié par le fait que les paquets peuvent arriver en désordre à la destination.

En remarquant que la réception d'un acquittement dupliqué, est dûe à un départ d'un paquet du réseau (un paquet est servi, il est arrivé à la destination) ; l'envoi d'un nouveau paquet à chaque réception d'un acquittement dupliqué est alors justifié. Ce mécanisme, introduit spécialement dans Reno, est le *Fast Recovery Mechanism*

[17]. Il propose donc de remplacer les paquets ayant quitté le réseau par d'autres (nouveaux paquets).

3 TCP Reno

Les principales innovations dans TCP Reno par rapport à TCP Tahoe sont le facteur de réduction de la fenêtre, dans le cas où une perte est détectée par acquittements dupliqués, et les actions à entreprendre à ce moment. En effet, dans Reno lors de la détection d'une perte par acquittements dupliqués, *seul* le paquet supposé être perdu, mais pas les suivants, est retransmis, et la phase de *congestion* au lieu du *slow start* est exécutée à nouveau :

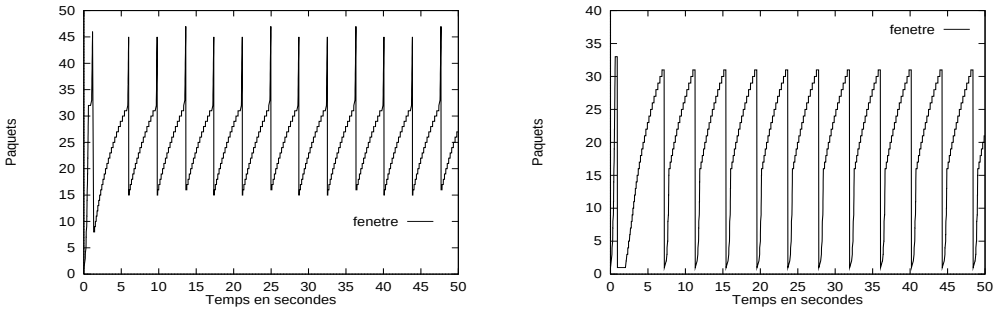
1. Le seuil du *slow start* (W_{th}) est mis à un demi de la taille actuelle de la fenêtre de congestion.
2. La fenêtre de congestion ($Cwnd$) est mise à W_{th} plus trois fois la taille d'un paquet (MSS, i.e. Maximum Segment Size).
3. A chaque réception d'un acquittement dupliqué, la source incrémente $Cwnd$ d'un paquet et transmet un nouveau paquet (si la nouvelle taille de $Cwnd$ le permet).
4. Quand le premier acquittement non dupliqué arrive, $Cwnd$ est mise à W_{th} (un nouveau cycle en phase de *congestion* est entamé).

Ces points constituent le mécanisme de *fast recovery* .

la figure 1, illustre les comportements de la fenêtre dans TCP-Reno et TCP-Tahoe. Notons que dans Reno, on cherche à éviter de recommencer la phase du *slow start* où les pertes sont souvent de l'ordre de la fenêtre (phase très agressive). Les lignes verticales que l'on constate après chaque perte, correspondent au mécanisme de *Fast Recovery* où la fenêtre augmente de la taille d'un paquet pour tout acquittement dupliqué reçu.

3.1 Reno vs Tahoe

Dans cet exemple, on peut facilement constater que Reno est meilleur que l'ancienne version, cependant ce n'est malheureusement pas toujours le cas. En effet, il existe des cas où les performances de Reno se dégradent de manière drastique.



(a) Reno (b) Tahoe

FIG. 1: Comportement de la fenêtre dans Reno et Tahoe

Ceci se produit quand plusieurs pertes successives ont lieu dans la même fenêtre de données (voir [13] pour une meilleure explication d'une fenêtre de données).

Cette situation peut se résumer de la façon suivante :
Dans le cas de pertes multiples (groupées), Reno réduit sa fenêtre de moitié à chaque détection d'une perte par acquittements dupliqués (il peut recouvrir, dans certains cas, jusqu'à trois pertes ce qui est rare). En l'absence d'acquittements (dupliqués), et si d'autres pertes restent encore non détectées, Reno attend l'expiration du timeout pour initialiser la phase du *slow start* et les recouvrir. Ceci peut prendre plusieurs délais d'aller retour, puisque la granularité du timeout est de 500 ms. La phase de *slow start* ainsi initialisée est entamée avec un seuil égal à la taille de la fenêtre au moment de détection de la première perte, divisée par 2^{1+n}, où n est le nombre de pertes (successives ou d'une même fenêtre de données [13]) détectées par acquittements dupliqués.

Dans le cas de Tahoe, quand de multiples pertes se produisent, dès la détection de la première perte la fenêtre est mise à un, et le seuil du *slow start* (W_{th}) est mis à la moitié de la taille de la fenêtre au moment où la perte est détectée. Par conséquent, Tahoe exécute la phase du *slow start* (et par la même occasion recouvre les autres pertes) plus rapidement que Reno, et avec un seuil (W_{th}) beaucoup plus grand. C'est la raison pour laquelle Tahoe est meilleur que Reno dans le cas de plusieurs pertes dans la même fenêtre de données (voir [23]).

Exemple

Supposons qu'il y ait trois pertes dans la même fenêtre de données. Supposons également que les deux premières pertes sont détectées par acquittements dupliqués et que la troisième par timeout. Reno initialise la phase du *slow start* (à la détection de la troisième perte) avec un seuil égal à la taille de la fenêtre, quand la première perte est détectée, divisée par 8 ($W_{th} = max\_win/8$) et ce pas avant la résolution du temporisateur (500 ms), au minimum. Dans la version Tahoe, la phase du *slow start* est initialisée avec un seuil quatre fois plus grand (max_win/2)), dès la détection de la première perte.

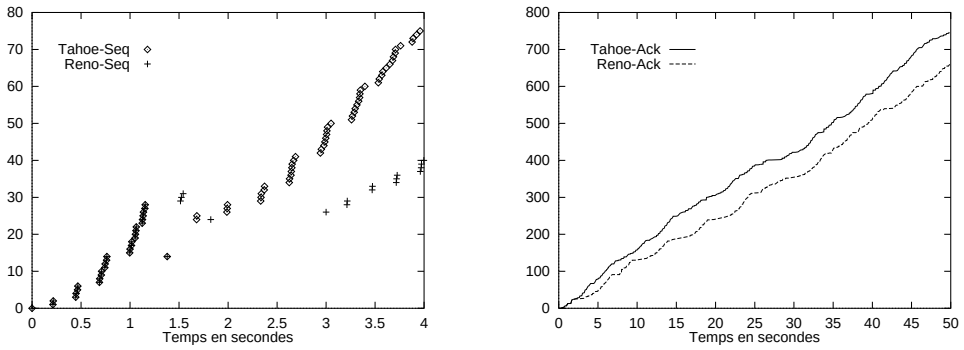


FIG. 2: Un exemple où Tahoe excède Reno

Remark 3.1 *Dans toutes les figures de cet article, l'axe des ordonnées correspond au numéro de séquence des paquets (Seq-) (resp (et/ou) des acquittements (Ack-)) et l'axe des abscisses aux instants de leurs transmissions (resp (et/ou) réceptions).*

La figure 2 illustre un exemple où Tahoe excède Reno en débit tout en ayant, relativement, le même taux de retransmission pour une durée de simulation de 50 sec. On peut constater dans la figure de gauche, où uniquement les premières quatre secondes figurent, que les séquences de transmission pour Reno et Tahoe (Seq sur la figure) sont les mêmes jusqu'à l'instant 1.37 où des acquittements dupliqués acquittant le paquet 13 arrivent.

Du coté de Reno, à l'arrivée de trois acquittements, le seuil du *slow start* est mis à $W_{th} = \lfloor 15/2 \rfloor = 7$ [1], car la fenêtre est de 15 paquets. Reno retransmet le paquet 14, et les paquets 29-31 dus au mécanisme de *fast recovery* (les "plus" sur la figure). A l'instant 1.80, trois acquittements dupliqués acquittant le paquet 23 sont reçus (dûs aux paquets 14, 29-31). Le paquet 24 est alors considéré perdu, et retransmis à l'occasion (ce dernier causera la réception de l'acquittement pour le paquet 25). Une nouvelle phase de *congestion* est alors entamée : le seuil du *slow start* est une nouvelle fois divisé par deux $W_{th} = \lfloor 7/2 \rfloor = 3$. En l'absence d'acquittements dupliqués (pour le paquet 25), on attend près d'une seconde et demi pour que le temporisateur expire. La phase du *slow start* est alors initialisée ($W_{th} = max(2, \lfloor 3/2 \rfloor = 2)$, ainsi on recouvre les autres pertes lentement, car la valeur du seuil du *slow start* est la plus petite possible.

Quant à Tahoe, dès la détection de la perte du paquet 14, par réception de trois acquittements dupliqués acquittant 13, il met le seuil du *slow start* à $W_{th} = \lfloor 15/2 \rfloor = 7$ et réinitialise la phase du *slow start* en mettant la fenêtre à 1. Il retransmet donc le paquet 14, puis à la réception de chaque acquittement non dupliqué, il (re)transmet les paquets subséquents. Il recouvre ainsi toutes les autres pertes à la même occasion, sans attendre l'expiration du timeout. Par conséquent, Tahoe réinitialise la phase du *slow start* avec un seuil presque quatre fois plus grand que Reno, et une seconde et demi plutôt.

Il serait tout de même injuste de ne pas souligner que des retransmissions inutiles ne sont pas évitées par Tahoe, ce qui peut alourdir le trafic dans les réseaux et causer des pertes à d'autres sources concurrentes.

3.2 Problème de "successive fast retransmits"

Un problème lié à l'exécution de plusieurs "fast retransmit" successifs, dans la même fenêtre de données, a été retracé dans [13]. Ce problème concerne essentiellement la version Tahoe, mais il peut avoir lieu dans Reno, aussi bien que dans Vegas, sous une autre forme (moins désastreuse), comme on le verra par la suite. Ce dysfonctionnement est dû à la réinitialisation de la phase du *slow start* à chaque fois qu'une perte est détectée ; cela cause la retransmission de tous les paquets, sans

[1] $\lfloor . \rfloor$ désigne la partie entière inférieure. Puisque on suppose que les paquets sont de taille fi xe, la fenêtre est alors un nombre entier de segments

qu'ils soient vraiment perdus (tous les paquets sont transmis deux fois voir figure 3).

La figure 3 en illustre un exemple. Le scénario est le suivant : la phase du *slow start* est exécutée initialement, et les paquets 0-83 sont transmis. Parmi ces paquets, six sont perdus (Perdu sur la figure) : 51, 53, 55, 57, 59, 61. A l'instant 3.27, la source reçoit des acquittements dupliqués acquittant le paquet 50 (figure de gauche). Le paquet 51 est alors retransmis et la phase du *slow start* est réexécutée. Les paquets 58-63 sont retransmis à l'instant 4.81, après avoir recouvert toutes les pertes précédant ces séquences.

A la réception du paquet 58 (doublement reçu), l'acquittement 58 est envoyé par la destination une nouvelle fois, puis deux fois l'acquittement 60 dûs à la réception des paquets 59 et 60 (seul 61 reste à recouvrir). Ainsi, dès la réception du paquet 61 par la destination, celle-ci envoie des acquittements dupliqués acquittant le paquet 83, tant que le paquet 84 n'est pas reçu (on recevra trois acquittements 83 dûs à la bonne réception des paquets 61-63).

C'est à ce moment où le problème apparaît. En effet, du coté de la source, à la réception des acquittements 58-60-60, les paquets 64-66 sont envoyés, la fenêtre étant de 7 (5 + deux acquittements distincts 60 et 83 ; *slow start*). Les paquets 84-90 sont envoyés dès la réception du premier acquittement du paquet 83, après quoi on reçoit encore trois acquittements dupliqués pour le paquet 83 dûs à la bonne réception des paquets 64-66, ce qui porte à six (6) le nombre de paquets dupliqués 83. Le paquet 84 est retransmis, en exécutant le "fast retransmit", puisque des acquittements dupliqués (83) sont reçus. Celui-ci est considéré à ce moment comme étant la plus grande en séquence transmise (la phase du *slow start* est recommencée).

A chaque réception d'un paquet (84-90) la destination les acquitte au fur et à mesure de leur arrivée. Ainsi, les paquets 84-90 seront retransmis à cause de leur propres acquittements. A l'arrivée de ces retransmissions à la destination, celle-ci envoie des acquittement dupliqués acquittant le paquet 90 (avant l'arrivée des autres paquets envoyés, dûs à l'ouverture exponentielle de la fenêtre : *slow start* 91-95). Finalement, le paquet 91 à son tour sera considéré perdu et retransmis, puis tous les autres à la même occasion et ainsi de suite, tous les paquets à partir du paquet 84 seront transmis deux fois sans qu'ils soient perdus, comme on peut le constater dans la figure de droite.

La solution proposée par S. Floyd [13] est de ne pas permettre un deuxième "fast retransmit" dans la même fenêtre de données. Ainsi, lors d'un premier "fast retrans-

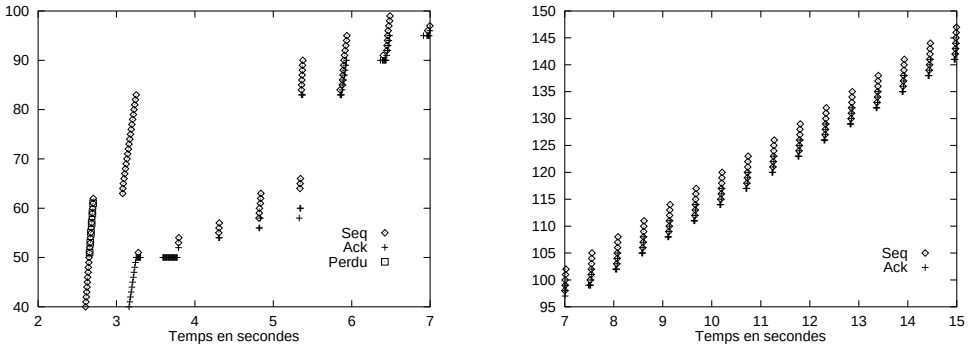


FIG. 3: Le problème de “fast retransmit” successifs dans Tahoe

mit”, Tahoe sauvegarde la plus grande séquence dernièrement transmise $snd\_high = snd\_nxt$, et un autre “fast retransmit” ne serait accepté que si le dernier acquittement est plus grand que snd_high.

Cette solution en effet évite le problème décrit précédemment, cependant quand plusieurs pertes ont lieu dans la même fenêtre, on aura forcement recours au temporisateur pour les recouvrir. Si on inclut cette solution également dans Reno, le premier problème (cf section 3.1) risque alors de surgir et Reno ne serait meilleur que Tahoe que dans le cas de pertes isolées ce qui est rare sur Internet.

Dans nos simulations de Vegas et Reno, nous n’avons pas inclus cette solution, car ce problème ne les concerne visiblement pas ; du moins comme on le verra plus tard (voir section 5.1), ses conséquences sont moins dramatiques que dans Tahoe. D’autre part, cette solution pénalise l’objectif de recouvrir de plusieurs pertes comme il sera discuté ultérieurement. Nous verrons également par la suite que la version New-Reno qui apporte des réponses aux pertes successives, peut souffrir également de ce problème.

4 TCP Vegas

L.S Brakmo et L.L Peterson [7] ont proposé un nouvel algorithme baptisé Vegas issue de quelques modifications du coté transmetteur (source) de Reno. Les modifications concernent essentiellement la façon de prédire la bande passante disponible, et la détection de pertes. Ce nouvel algorithme, selon ses auteurs [7], est capa-

ble d'interagir avec toute implémentation valide de TCP. Il permet, en particulier, d'améliorer de 37% à 71% le débit par rapport à Reno et de retransmettre entre un cinquième et la moitié de ce que retransmet Reno. Dans [1], les auteurs confirment ces améliorations, en utilisant un émulateur de réseaux. Ils rapportent un gain en débit d'au moins 4%-20% par rapport à *Reno* pendant qu'il retransmet le même taux donné dans [7].

Les mécanismes de "fast retransmit" et "fast recovery" y sont également implémentés. Un nouveau *slow start* est introduit visant a minimiser les pertes durant cette phase, où elles peuvent être très importantes. La taille de la fenêtre de congestion est ajustée selon la phase exécutée :

Phase de Congestion

Dans sa phase de *congestion*, Vegas estime la quantité de données sauvegardées dans les files d'attente intermédiaires et décide d'augmenter ou de réduire la taille de la fenêtre en s'y référant. Cette estimation est faite une fois durant chaque délai d'aller retour (RTT) : Vegas initialise $BaseRTT$ au minimum de tous les délais d'aller retour (round trip time) mesurés (y compris ceux des précédents RTT) et calcule le débit $Expected$ (débit maximum) comme étant $Expected = Cwnd/BaseRTT$ où $Cwnd$ est le nombre d'octets en transit (i.e. la taille de la fenêtre). Pendant cette période, Vegas enregistre l'instant de transmission d'un paquet distingué et le nombre d'octets transmis entre cet instant et l'instant d'arrivée de son acquittement, et calcule ainsi le débit réel de transmission comme étant $Actual = Cwnd/RTT_d$, où RTT_d est le délai d'aller retour du paquet distingué.

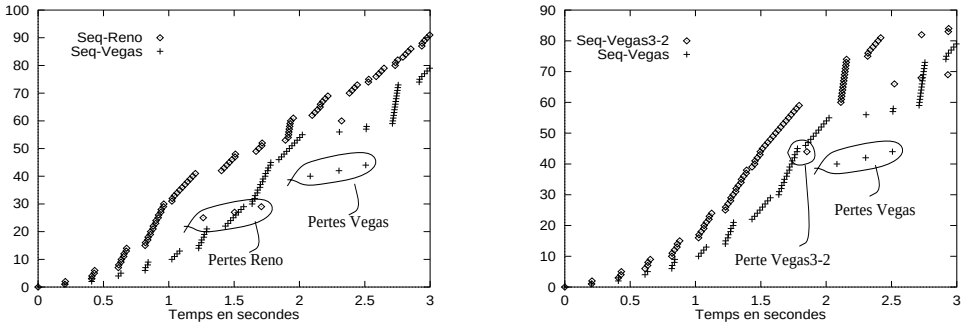
La source (transmetteur) définit trois seuils α, β et γ (dans nos simulations, nous considérons $\alpha = 2$, $\beta = 4$ et $\gamma = 1$ ($Vegas_{2,4}$)). Si $Expected - Actual \leq \alpha/BaseRTT$ alors la fenêtre est augmentée de $1/Cwnd$ pour chaque acquittement reçu (comme Reno). Cependant, si $Expected - Actual > \beta/BaseRTT$, la fenêtre est décrémentée d'un paquet. Enfin, si la différence $Expected - Actual$ est comprise entre les deux seuils (α et β), la fenêtre est maintenue inchangée.

A noter que dans la version implémentée (Vegas release 0.8 [6]), RTT_d est considéré comme la moyenne des délais d'aller retour, prise sur l'ensemble des paquets acquittés entre la transmission du paquet distingué et la réception de son acquittement (Ack).

Phase du *Slow Start*

Le principe est le même que dans le cas de Reno, à la seule différence que Vegas double la taille de sa fenêtre un RTT sur deux, contrairement à chaque RTT dans le cas de Reno. La phase du slow start prend fin une fois que Vegas détecte qu'une file se construit. Ceci est supposé être vrai dès que la fenêtre atteigne une certaine valeur W_{th} pour laquelle la différence $Expected - Actual$ dépasse un certain seuil $\gamma/BaseRTT$, à ce moment Vegas bascule vers sa phase de *congestion*. Cette technique peut permettre d'éviter des pertes pendant la phase du *slow start*, où elles peuvent être de l'ordre de la moitié de la fenêtre dans le cas de Reno.

4.1 Slow Start dans TCP Vegas et TCP Reno



(a) Slow start dnas Reno et Vegas

(b) Slow start dans Vegas et Vegas3-2

FIG. 4: *slow start* dans Vegas et Reno

Le fait de doubler la fenêtre une fois tous les deux RTT ne change rien de l'agressivité de la phase du *slow start*. Cette phase, dans beaucoup de cas, prend uniquement plus de temps que Reno et cause exactement les même pertes que ce dernier. En effet, Vegas met entre 0 et 75 % de temps en plus pour transmettre le même nombre de segments que Reno durant cette phase [2]. En s'appuyant sur des simulations, on remarque que les pertes durant cette phase dépendent essentiellement du délai d'aller retour et de la capacité des files d'attente aux routeurs. Quand les délais d'aller retour sont suffisamment grands, les files intermédiaires de petites capacités et le trafic exogène est faible, la taille de la fenêtre pour laquelle les pertes

se produisent dans Reno est, souvent, la même que dans Vegas. De plus, le nombre de pertes est également le même.

Par contre, sous les mêmes hypothèses, si au lieu de doubler la fenêtre une fois sur deux RTT on envoie trois paquets à la réception de deux acquittement, on obtient des performances meilleures que celles de Vegas. Autrement dit, durant la phase du *slow start* on envoie un paquet pour le premier acquittement et deux paquets pour le second. La phase du *slow start* ainsi obtenue est plus rapide que celle de Vegas et permet de meilleures performances.

la figure 4 (a) illustre un exemple assez simple où une source envoie des segments vers une destination à travers un routeur qu'elle utilise seule (sans trafic exogène). Cet exemple montre qu'on améliore en rien les pertes dans Vegas par rapport à Reno. En effet, on peut remarquer que les pertes ont lieu quand la fenêtre atteint la taille de seize segments pour la première fois dans les deux cas (Reno et Vegas), et le nombre de pertes constatées, dans la même fenêtre, est le même (trois). Ainsi, on peut dire que dans ces situations, hormis le mécanisme de détection de la bande passante dans Vegas durant la phase du *slow start*, il n'y a aucune amélioration par rapport à Reno. Vegas ne fait que retarder les pertes.

La figure 4 (b) quant à elle, illustre un exemple du *slow start* dans Vegas et le *slow start* modifié (au lieu de doubler la fenêtre une fois sur deux RTT on envoie deux paquets pour un acquittement une fois sur deux). D'une part, on peut voir qu'il ne se produit qu'une seule perte par fenêtre. En fait, cela dépend également de la capacité de la file d'attente au niveau du routeur, mais le nombre de pertes est toujours moins important que dans Vegas ou Reno.

La conclusion à tirer de ces constatations est que les pertes durant la phase du *slow start* dépendent essentiellement de la capacité des files d'attente et du seuil du *slow start*. Elle dépendent moins de la bande passante disponible. Si on choisit le bon seuil du *slow start* dans Reno, alors on aurait de meilleures performances que Vegas durant cette phase pour des délais assez grands. D'autres exemples analytiques pour analyser ce cas sont donnés dans [2].

Détection de pertes

Dans ce qui suit, nous décrirons le mécanisme de retransmission de Vegas et discuterons les différentes techniques utilisées pour détecter les pertes.

Vegas élargie le mécanisme de retransmission de Reno comme suit : Vegas lit et sauvegarde l'instant de transmission de chaque paquet. Quand un acquittement est reçu, Vegas lit l'heure et calcule ainsi le délai d'aller retour moyen (smooth RTT), la déviation moyenne, et le timeout à granularité fine (fine grain timeout (*v_timeout*)). Ce timeout est calculé de la même manière que Reno avec une granularité très fine, non pas un multiple de 500 ms. Le timeout à grosse granularité (coarse timeout) de Reno reste effectif sous Vegas. Vegas décide de retransmettre un paquet dans les situations suivantes :

– A la réception d'un acquittement dupliqué, Vegas vérifie si le temps écoulé depuis la transmission du premier (en séquence) paquet non encore acquitté est plus grand que le timeout à granularité fine (v_timeout). Si c'est le cas, le paquet est alors considéré perdu et retransmis sans attendre la réception du troisième acquittement dupliqué (requis par Reno).

Cette technique parfois permet de détecter des pertes plus rapidement à cause du fait que le temps de service des paquets n'est pas négligeable, et les capacités des files d'attente au niveau des routeurs tendent à être de plus en plus importantes. Ce n'est pourtant pas l'avis de V. Jacobson [20] qui estime que très souvent le mécanisme en question échoue. Les arguments avancés étant que le timeout (v_timeout) doit être au moins égale à 2RTT, au moment où le temps entre le premier et le troisième acquittement dupliqué est typiquement très petit. En d'autre termes, si perte il y a, le temps entre l'acquittement du paquet bien reçu et le troisième acquittement dupliqué ne dépasse pas souvent le RTT.

– A la réception du premier ou du deuxième acquittement après une retransmission dûe aux acquittements dupliqués, Vegas vérifie encore une fois si le temps écoulé depuis la transmission du premier paquet non encore acquitté dépasse la valeur du timeout de Vegas (v_timeout). Si c'est le cas, alors Vegas le retransmet.

Cette technique permet de détecter les pertes plus rapidement et de recouvrir les pertes groupées (multiples), sans réinitialiser la phase du *slow start*. Par conséquent, cela permet de palier au problème dont souffre la version Reno, notamment plusieurs pertes dans la même fenêtre de données, comme expliqué précédemment. Cette idée est reprise dans TCP New-Reno, comme on le verra par la suite, sans toutefois que la vérification du temps écoulé depuis la transmission soit faite. En effet, New-Reno retransmet dès la réception d'un acquittement non dupliqué acquittant

des séquences inférieures à la plus grande séquence transmise (non seulement le premier et le deuxième acquittement non dupliqué après une retransmission).

Les performances de Vegas semblent en effet dépasser celles de Reno. D'autant plus que cette dernière version est moins agressive et cause moins de pertes aux trafics concurrents. Cependant, elles reposent essentiellement sur l'estimation du délai d'aller retour [20]. Si ce délai est mal estimé, on risque soit de saturer le réseau ou de sous estimer la bande passante disponible, ne serait-ce que pour un moment assez court. Ajouter à cela que les performances de Vegas se dégradent en fonction du nombre de connexions [2].

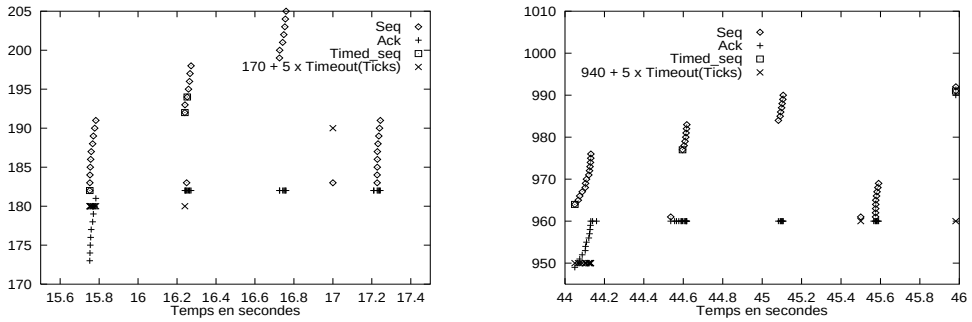
Parmi les autres désavantages à mettre au compte de cette version, le temps que Vegas met pour vérifier et mettre à jour les différents champs concernant les dates d'envoi des paquets [20] (overhead). Ce desavantage est mineur, du moment que cet excès de temps est de l'ordre de quelques dizaines d'operations par seconde [1].

5 Problèmes dans TCP Reno (Vegas) : vers New-Reno

Les mécanismes de "Fast retransmit" et "Fast recovery", sont connus pour être efficaces dans l'amélioration du débit et la détection rapide des pertes. Toutefois, comme on l'a souligné ci-dessus, ils peuvent être la source de certains dysfonctionnements, et par conséquent causent quelques problèmes. En plus des problèmes relatés ci-dessus, à savoir le problème des "fast retransmit" successifs (successive fast retransmit) et les pertes multiples, nous allons voir dans ce qui suit d'autres problèmes qui surgissent dans Vegas et Reno. Le premier problème concerne l'exécution d'un faux "fast retransmit" suivis d'un faux "fast recovery", après un timeout. Il se produit souvent quand les délais d'aller retour sont de l'ordre de grandeur de la résolution du temporisateur (500 ms) et que les capacités des files d'attente dans les routeurs (dans le réseau) sont suffisamment larges [3]. Le deuxième concerne l'exécution d'un faux "fast retransmit" [14], qui peut être vu comme cas particulier du premier.

5.1 Problème des faux “fast (retransmit) recovery” après time-out

Ce problème apparaît dans Vegas et Reno, et se traduit par la transmission inutile de plusieurs paquets. Il peut se produire quand on exécute le mécanisme de *fast recovery* (fast retransmit) après l’expiration du temporisateur de retransmission. Un problème similaire “false fast retransmit” est rapporté dans [14] voir section 5.2 ; c’est également un cas particulier des “fast retransmit” successifs. Un exemple est illustré dans la figure 5 (Vegas), où les instants auxquels les paquets sont transmis et ceux auxquels leurs acquittements arrivent sont représentés. Dans la figure, les acquittements sont représentés par des "plus", les instants de mise à jour du timeout sont représentés par des "X", les séquences des paquets transmis par des "diamants" et enfin, les paquets temporisés pour l’estimation du RTT par des "carrés". Les acquittements ayant la même séquence correspondent à des acquittements dupliqués indiquant qu’une perte a lieu.



(a) Vegas (b) Reno

FIG. 5: Problème du “fast recovery” après timeout à grosse granularité (Reno)

Le scénario est le suivant (figure 5 (b)) : à l’instant 44.53 la source reçoit trois acquittements (Acks) dupliqués acquittant le paquet 960 ; ce qui traduit la perte du paquet 961, qui est, par conséquent, retransmis après avoir réduit la fenêtre à 11 ($\lfloor 16/2 \rfloor + 3$). Tant que des Acks dupliqués (acquittant le paquet 961) arrivent, la taille de la fenêtre augmente à raison d’un paquet par Ack reçu (fast recovery) et un nouveau paquet est transmis si la taille actuelle de la fenêtre le permet. Les paquets 977-983 seront ainsi transmis (977 était le dernier paquet transmis avant

la réception du premier acquittement dupliqué). Le paquet 961 est une deuxième fois perdu. A l'instant 45.08, on reçoit encore des acquittements dupliqués dûs à la bonne réception des paquets 977-983, et les paquets 984-990 sont alors transmis (toujours dans la phase de "fast recovery").

A l'instant 45.500 le temporisateur expire, la phase du *slow start* est entamée en réinitialisant pratiquement tous les paramètres : la plus grande séquence transmise est considérée comme étant le premier paquet non acquitté, le paquet 961 en l'occurrence, la fenêtre est remise à 1, le seuil du *slow start* à [30/2] = 15, car la fenêtre atteint 30 à cause du mécanisme de fast recovery, et enfin le nombre d'acquittements dupliqués reçus est remis à zéro. Le paquet 961 est alors retransmis.

Les acquittements des paquets 984-990, qui étaient bien reçus arrivent juste après l'expiration du temporisateur. A la réception de trois d'entre eux (7 acquittements en tout), le paquet 961 est considéré perdu (faux "fast retransmit") et retransmis à tort. A ce moment, le seuil du *slow start* est mis à $ssthresh = max(2, [Cwnd/2]) = 2$, la fenêtre, quant à elle, atteint 5 ($ssthresh + 3$ acquittements dupliqués) et les paquets 962-965 sont ainsi (re)transmis (fast recovery). La source recevra encore quatre acquittements dupliqués. Dû au mécanisme de "fast recovery", les paquets 966-969 seront retransmis eux aussi à tort.

On remarque que les paquets 961-969 sont retransmis sans qu'ils soient perdus pour autant, et ce à cause de l'exécution du "fast retransmit" suivis d'un "fast recovery" qui ne devrait pas avoir lieu.

5.1.1 Solution

La dernière retransmission du paquet 961 peut être évitée en initialisant le nombre d'acquittements dupliqués reçus à trois juste après l'expiration du timeout à grosse granularité (ceci peut se traduire dans le code de TCP (TCP Vegas release 0.8 [6] ou dans BSD.4.3) par $tp \rightarrow t\_dupacks = tp \rightarrow t\_rexmtthresh$ dans (v)tcp_timer.c au lieu de $tp \rightarrow t\_dupacks = 0$).

La solution donnée par S. Floyd [13] comme remède au problème de "fast retransmit" successifs ne peut être solution à ce problème. Même si elle peut éviter la retransmission du paquet 961 pour la quatrième fois, il n'en demeure pas moins que les autres paquets seront retransmis du moment que les paramètres sont pratiquement tous réinitialisés (c'est le premier paquet en séquence non encore acquitté qui

est considéré comme étant la plus grande séquence transmise). D'autre part, cette solution n'est pas envisageable pour les raisons citées ci-dessus (cf section 3.1).

Le problème de la retransmission des paquets 962-969 peut être résolu en évitant d'exécuter le mécanisme de "fast recovery" après l'expiration du temporisateur de retransmission (i.e. timeout à grosse granularité). Ceci peut se faire en désactivant un indicateur (fast_recovery) quand le timeout expire, et l'activant une fois qu'un acquittement non dupliqué soit reçu (ceci peut se traduire dans le code TCP (TCP Vegas release 0.8 [6] ou BSD.4.3) par l'incrémentation de la fenêtre uniquement si ($tp \rightarrow t\_dupacks > tp \rightarrow t\_rexmtthresh$ && fast_recovery)).

Une deuxième solution que nous proposons consiste simplement à incrémenter la fenêtre à chaque réception d'un acquittement dupliqué, uniquement si la fenêtre actuelle est plus large que le seuil du *slow start*. Cette solution est envisageable du moment que le fast recovery suit toujours le "fast retransmit". Ainsi, si on ne permet pas de fast retransmit, la fenêtre dans ce genre de situation sera toujours inférieure au seuil du *slow start*.

Sous Vegas, ce problème peut également apparaître. On peut avoir exactement le même comportement, si on considère que le scénario est le même pour Vegas jusqu'à l'instant 44.53 (b). La figure 5 (a) en illustre un autre exemple pour Vegas.

Il faut tout de même rappeler que la situation retracée dans cette section ne se produit pas si les délais d'aller retour sont suffisamment petits, et que la capacité de stockage à la destination est limitée. A ce moment, ce qui risque de se produire réellement, est que la source continue d'envoyer de nouveaux paquets, dûs au fast recovery, et de recevoir des acquittements dupliqués pour le paquet 960, jusqu'à ce que la fenêtre soit réduite à zéro, à cause de la capacité de stockage au niveau du récepteur. Ainsi, la source rentre dans l'état "persist", enclenché chaque fois que la fenêtre se réduit à zéro, pour une autre raison que celle pour laquelle est prévu cet état [29, 30].

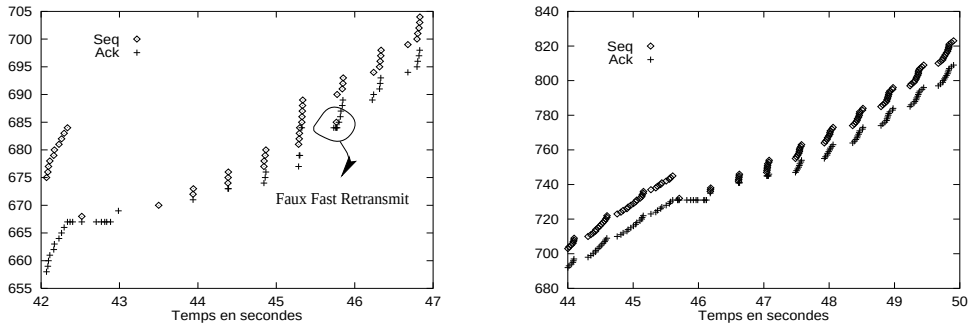
Un paquet de taille d'un octet (i.e. formant un datagramme IP de 41 octets) est à ce moment envoyé, et le premier (en séquence) paquet non encore acquitté est considéré comme étant la plus grande séquence envoyée. Le problème pourrait être résolu une fois que tout le segment manquant est reçu par la destination, octet par octet, ce qui prendrait un temps énorme (au minimum deux fois le produit taille d'un segment en octet fois le délai d'aller retour). Ceci se produit dans le cas où la destination acquitte les octets un par un. Cependant, la solution de Clark [8, 30], au problème du **syndrome de la fenêtre stupide** (*silly window syndrome*) empêche

une destination d'acquitter octet par octet, du moment que la capacité de stockage à la destination n'a pas changé. Par conséquent, dans notre cas la source retransmet le même octet à chaque expiration du temporisateur de "persistance", et une situation d'**interblocage** (*deadlock*) apparaît. La source demeurera ainsi, dans cet état jusqu'à l'avortement de la connexion, bien que la communication aurait pu se dérouler de la plus belle façon !. Ce qui constitue un problème de plus.

On pourrait penser que ce problème de "faux fast (retransmit) recovery" est simplement un problème potentiel dans TCP. Toutefois, beaucoup d'articles suggèrent que la résolution du temporisateur soit réduite pour améliorer le débit dans les réseaux à haut débit (ATM par exemple). Ajouter à cela que les liens satellites peuvent aisément engendrer des délais de l'ordre de 300 ms, ainsi avec une résolution de 300 ms (existante), on peut s'attendre à rencontrer ce genre de dysfonctionnements.

5.2 Le problème du faux "fast retransmit" : New-Reno

Un problème similaire à celui retracé ci-haut dans Reno est décrit dans [14]. La solution proposée pour y palier a donné naissance à une version de TCP appelée TCP New-Reno. En réalité cette version se rapproche beaucoup plus de Tahoe que de Reno, puisque à chaque détection d'une perte, une "partie" de la phase du *slow start* est exécutée.



(a) Reno (b) New-Reno

FIG. 6: Problème du faux "fast retransmit" dans Reno

Le problème du faux "fast retransmit" est retracé dans la figure 6 où figurent les séquences des paquets et de leurs acquittements. Dans cette figure, le scénario

est le suivant : Après un timeout à l'instant 43.5, Reno déclenche la phase du *slow start* et retransmet le paquet 670, qui causera la réception de l'acquittement 671. Les paquets 672-673 sont alors (re)transmis, et causeront à leur tour la réception de deux acquittements dupliqués pour le paquet 673, et les paquets 674-676 sont ainsi retransmis (la fenêtre est à 3 et le seuil du *slow start* à 4). A l'instant 44.84, dûs aux acquittements des paquets 674-676, les paquets 677-680 sont retransmis.

L'acquittement 677 (678 était perdu lors de sa première transmission), et deux acquittements dupliqués pour 679 sont renvoyés par la destination à la réception des paquets 677-679 (le paquet 680 était perdu lors de sa première transmission). Ainsi, les paquets 681-684 sont (re)transmis (il étaient bien reçus lors de leurs première transmission) avant la réception de l'acquittement 984 dû à la bonne réception du paquet 980 retransmis. A cet instant, la source transmet les paquets 985-989. A l'arrivée des paquets 981-984 à la destination, celle-ci renvoie encore des acquittements dupliqués pour le paquet 984, qui sera une fois de plus considéré perdu à cause des trois acquittements dupliqués, et par la suite retransmis inutilement.

Le comportement qu'on souhaiterait avoir idéalement dans ce cas de figure, est retracé dans la figure 6 (b). C'est en fait le comportement souhaité de New-Reno ; la nouvelle version conçue pour répondre à ce genre de problèmes. Nous verrons dans ce qui suit que cette version comme toutes les autres a sa part de dysfonctionnements.

New-Reno est proposé non seulement pour palier au problème du " faux fast retransmit", mais aussi pour améliorer les performance de Reno dans le cas de pertes multiples. L'idée est de s'inspirer du fonctionnement de Tahoe puisque ce dernier recouvre nettement mieux que Reno les pertes multiples dans la même fenêtre de données.

Il s'agit donc d'utiliser le *slow start*, tout comme Tahoe, lors d'une détection de pertes par acquittements dupliqués, et de recommencer la phase de *congestion* une fois que toute la fenêtre est recouverte (toutes les données d'une même fenêtre de données sont acquittées).

Les changements apportés à Reno sont résumés dans le diagramme 7 où l'essentiel des modifications concernent le mécanisme de "fast retransmit".

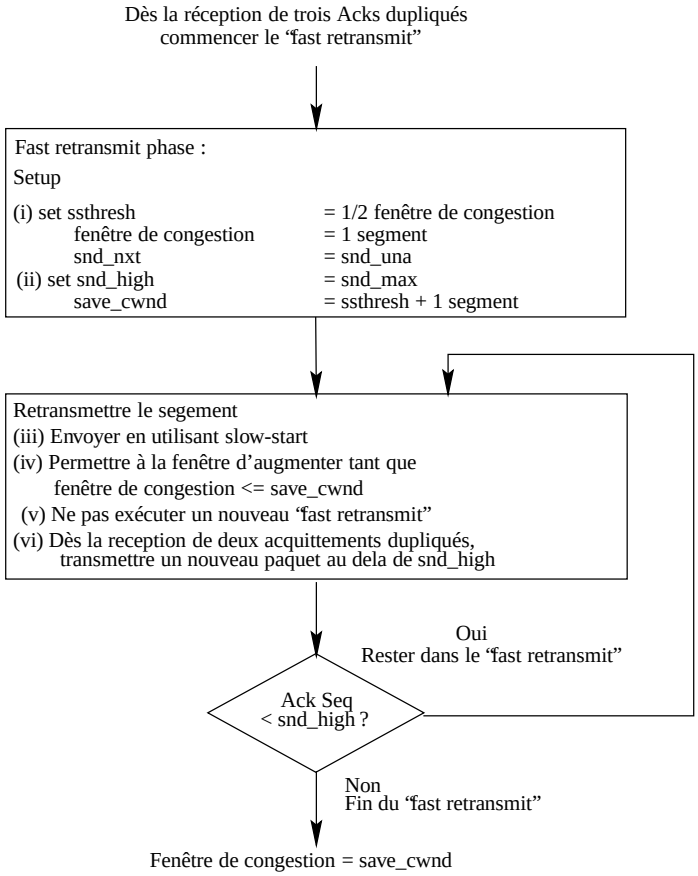


FIG. 7: “fast retransmit” dans New-Reno

5.3 New-Reno et problème des “fast retransmit” successifs

Nous avons simulé la nouvelle version de TCP (New-Reno) [14], et essayé de déceler les dysfonctionnements qui pouvaient surgir. Nous avons utilisé le même modèle que précédemment. Nous avons constaté que si on considère que la phase de “fast retransmit” se termine dès la réception de l’acquittement du plus grand en séquence, snd_high en l’occurrence (tel est le cas dans [14] pp. 276), alors le problème des “fast retransmit” successifs décrit dans la section 3.2 pour Tahoe, réapparaît dans cette version.

La figure 8 en illustre un exemple. Dans cette figure, les séquences des paquets transmis sont représentés par des "diamants", leurs acquittements par des "plus" et enfin les séquences qui doivent être acquittées avant de permettre un nouveau “fast retransmit” sont représentées par des "carrés". La phase (vi) dans le diagramme de New-Reno (figure 7) n’est pas prise en compte dans cet exemple, ainsi on ne permet pas de “fast recovery”. Cela ne change rien à l’apparition de ce problème.

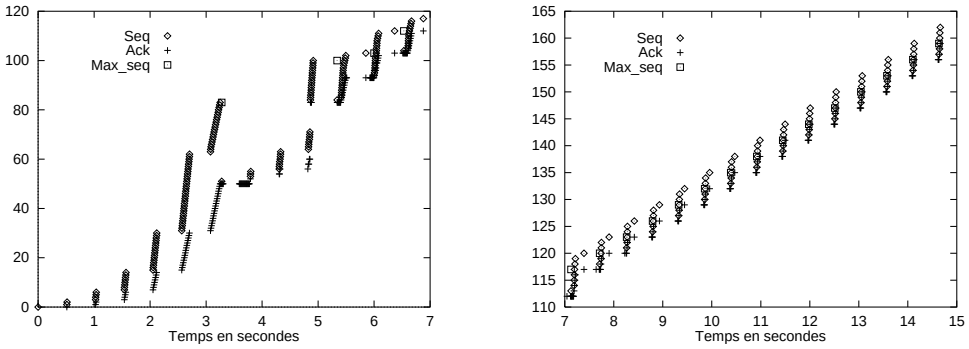


FIG. 8: Problème de “fast retransmit” successifs dans New-Reno

5.3.1 Solutions et Remarques

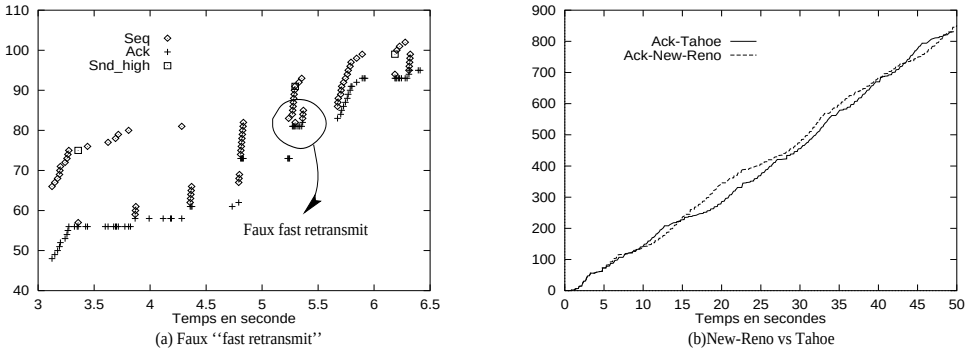
Le problème disparaît par contre si la fin de la phase de “fast retransmit” se termine, non pas une fois que snd_high est acquitté ($Ack \geq snd\_high$), mais quand le segment envoyé après snd_high est acquitté ($Ack > snd\_high$).

Le problème de retransmissions inutiles n’est toutefois pas résolu dans ce dernier cas, si le “fast recovery” est permis (phase (vi) dans le diagramme 7). En effet, la figure 9 (a) illustre un exemple où plusieurs paquets sont retransmis inutilement, dûs essentiellement aux acquittements des paquets transmis durant la phase de “fast recovery”. C’est pour cette raison, d’ailleurs, que le fast recovery dans Tahoe risque d’engendrer des situations catastrophiques.

Finalement, avec l’inclusion du “fast recovery”, on peut remarquer que le problème du “faux fast retransmit” n’est pas résolu non plus. Le cercle dans la figure 9 (a) en illustre un, et pire encore on retransmet plus d’un paquet inutilement.

Mettre à jour snd_high à chaque fois qu’un paquet nouveau (au delà de snd_max) est envoyé (de telle sorte que $snd\_high = snd\_max$), peut paraître une solution per-

mettant de résoudre ce problème. Cependant, cette solution risque de "bloquer" la fenêtre et se trouver indéfiniment dans le *slow start*. La fenêtre risque également d'atteindre $save\_cwnd$ et rester à cette valeur, envoyant ainsi avec une fenêtre restreinte (constante).



(a) Faux ''fast retransmit'' (b)New-Reno vs Tahoe

FIG. 9: Faux "fast retransmit" dans New-Reno (a), et comparaison avec Tahoe (b)

Finalement, la seule solution qui nous paraît raisonnable est de "se passer" du mécanisme de "fast recovery". Ainsi, on est presque certain que tous les problèmes cités ci-dessus n'apparaîtront plus. Au moins, dans de nombreuses simulations que nous avons faites avec cette version "modifiée", aucun de ces problèmes n'est rencontré. La version de New-Reno ainsi obtenue est la même que celle décrite dans le diagramme 7, sans l'étape (vi) ("fast recovery") et on test sur $Ack \leq snd\_high$ au lieu de $Ack < snd\_high$.

Si on inclut la solution au problème de "fast retransmit" successifs proposée par S. Floyd [13], il est facile de remarquer que finalement, la version New-Reno qu'on obtient est à notre avis plus que jamais proche de Tahoe que de Reno. La seule différence entre cette version et Tahoe est que cette dernière oublie de remettre la fenêtre à $ssthreh + 1$ une fois que la plus grande séquence dans la fenêtre est acquittée, et permet également à la fenêtre d'augmenter au delà de $save\_cwnd$.

Par conséquent, il est évident que dans beaucoup de cas, l'ancienne version de Reno dépasse cette nouvelle version en débit. Les mêmes études comparatives entre Reno et Tahoe pourraient être valables pour New-Reno et Reno. La figure 9 (b) illustre un exemple où New-Reno et Tahoe ont pratiquement la même allure et permettent le même débit de transmission.

5.4 Remarques et Propositions

Dans ce qui suit, nous proposons quelques changements à Reno, pour obtenir quelques améliorations. Ces changements s'inspirent de Vegas et concernent essentiellement la détection de perte. On peut penser à introduire exactement les mêmes techniques de détection de pertes de Vegas dans Reno pour recouvrir des pertes multiples. Cependant, ces techniques demandent un peu de temps d'exécution, bien que ça soit négligeable, nous préférons les utiliser autrement.

Dans [14], comme dans beaucoup de papiers, on insiste sur le fait que le seuil du *slow start* doit être minutieusement choisi pour éviter les pertes multiples. Dans Vegas, on propose d'estimer la bande passante disponible en calculant le RTT exact de chaque paquet transmis. Cette dernière solution peut être retenue, si on fait abstraction du temps dépensé dans le calcul des différents débits (débit_actuel, débit_attendu ...etc. Voir section 4).

Une autre méthode consiste à estimer la bande passante disponible (ou produit bande passante délai), dès l'établissement de la connexion, et ce par la technique de "packet pair" [24]. En considérant que le délai d'aller retour est facilement mesurable, on peut avoir une estimation de la bande passante en envoyant deux paquets consécutifs. A la réception de leurs acquittements respectifs, le temps entre ces deux acquittements peut donner des indications sur le temps de service dans le réseau. C'est une méthode qui peut donner de bons résultats, si les changements de la densité du trafic sont suffisamment limités. Par contre, elle peut facilement engendrer une surestimation ou une sous-estimation des capacités du réseau, si le trafic est très variable, mais tout cela est une autre histoire !.

On a vu dans New-Reno que l'objectif est de retransmettre tous les paquets tant que la plus grande séquence n'est pas encore acquittée, en utilisant le *slow start*. Dans le cas où la première retransmission par "fast retransmit" est perdue une seconde fois, toutes les versions (Tahoe, Reno, Vegas, New-Reno) doivent attendre l'expiration du timeout pour la recouvrir. Ainsi, New-Reno (et les autres) suppose qu'une retransmission est toujours bien reçue. On peut se demander, à ce moment, à quoi pourrait servir l'exécution d'un *slow start* après un "fast retransmit" ?.

Il est vrai que l'exécution d'un *slow start* permettrait de recouvrir de plusieurs pertes au même temps, si les pertes ont lieu de façon groupées. Mais il est également vrai que si les pertes sont dispersées (des pertes durant la phase de *congestion* par exemple), alors beaucoup de données bien reçues par la destination seraient retrans-

mises inutilement (problème très connu de Tahoe). Par conséquent, l'agressivité de la phase du *slow start* risque, fort bien, d'engendrer de nouvelles pertes et même la perte des données qu'on essaye de recouvrir. L'interminable attente de l'expiration du timeout serait alors inévitable, du moment qu'un nouveau "fast retransmit" dans la même fenêtre de données n'est pas permis.

5.4.1 Proposition

Puisque New-Reno (et les autres) suppose qu'une retransmission est toujours bien reçue, on pourrait alors envisager de retransmettre uniquement les paquets jugés susceptibles d'être perdus. Cela peut se faire en retransmettant un paquet (ou un nombre limité de paquets successifs) à la réception d'un acquittement non dupliqué après une retransmission (technique utilisées dans Vegas).

Dans le cas de retransmission d'un unique paquet, le recouvrement de plusieurs pertes se fera durant un nombre de RTT égale au nombre de pertes. Cela peut constituer un défaut ; cependant, les délais d'aller retour tendent de plus en plus à être très petit (fibre optique) et peuvent être négligés par rapport à la résolution du temporisateur. A rappeler également que le "faux fast retransmit" dans New-Reno est dû aux multiples retransmissions durant le "fast retransmit", en exécutant la phase du *slow start*. C'est la raison essentielle qui justifie le choix de ne retransmettre qu'un seul paquet à la réception d'un acquittement non dupliqué, après une retransmission.

En incluant les solutions de la section 5.1, pour éviter le problème du "faux fast recovery", ainsi que la retransmission à la réception d'un acquittement non dupliqué durant un "fast retransmit", on peut raisonnablement améliorer Reno. En effet, la version ainsi obtenue n'empêche pas de "fast recovery" dont l'utilité est tout à fait justifiée.

Les légères modifications apportées à Reno sont décrites dans le diagramme 10. Ce diagramme montre l'essentiel des opérations à effectuer à chaque réception d'un acquittement, lors de l'exécution d'une phase de "fast retransmit". Cette dernière commence dès la retransmission d'un paquet, engendrée par la réception de trois acquittements dupliqués, jusqu'à ce qu'un segment plus grand que la plus grande séquence transmise avant cette retransmission, soit acquitté.

A remarquer que lors de la retransmission d'un paquet dûe à la réception d'un acquittement non dupliqué durant la phase de "fast retransmit", la fenêtre reste in-

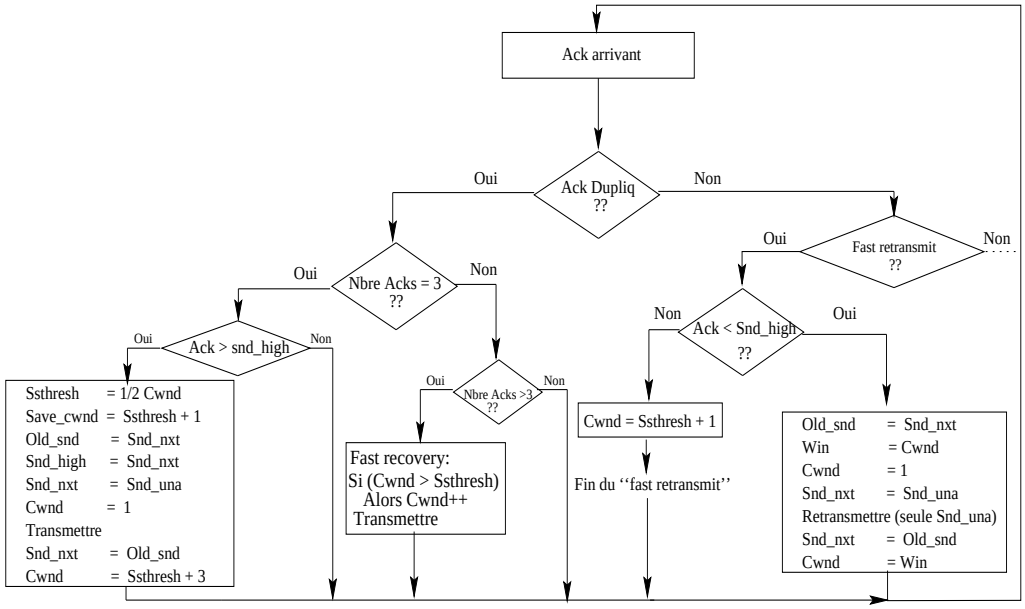


FIG. 10: Diagramme de Reno modifiée lors de la réception d'un Ack dans la phase de "fast retransmit"

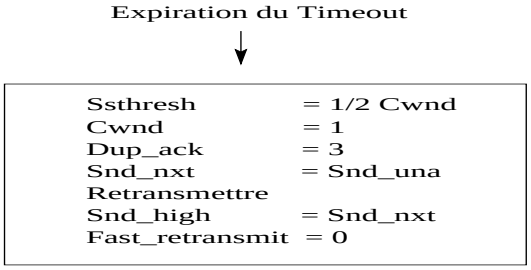


FIG. 11: Les opérations à effectuer lors de l'expiration du timeout

changée. Ceci, pour éviter le problème de réduction de la fenêtre dans Reno lors de multiples pertes (cf section 3.1).

Les opérations à entreprendre quand le timeout expire, sont retracées dans la figure 11 (solution au problème des "faux fast recovery". Voir section 5.1.1) :

On remarquera que les changements ne sont pas substantiels et qu'uniquement le transmetteur est concerné. Les améliorations par rapport à TCP-Reno sont indiscutables, et tous les problèmes décrits le long de ce travail ne figurent pas dans cette dernière version de Reno. Cette version de TCP est très proche de TCP Vegas, excepté que Vegas essaye d'estimer la bande passante disponible. Ce qui est vraiment apporté de nouveau dans cette version est la retransmission à la réception d'un acquittement non dupliqué après une première retransmission. Une version similaire à celle décrite ici a été analysée à l'aide des simulations par K. Fall et S. Floyd [12].

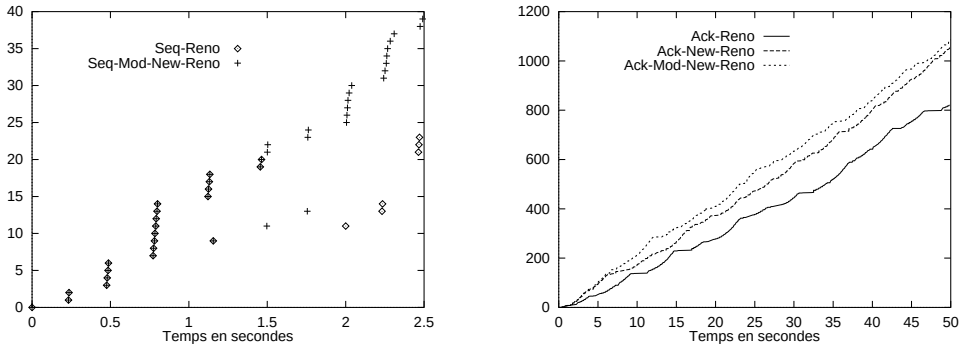


FIG. 12: Reno, New-Reno et les modifications apportées à New-Reno

La figure 12 (a) illustre l'exemple précédent où Tahoe et New-Reno ont pratiquement le même débit de transmission. On peut remarquer que le New-Reno modifié est plus performant que New-Reno. En effet, dans l'exemple en question, plusieurs pertes ont eu lieu (trois). Reno recouvre d'une première, après la réception de trois acquittements dupliqués ; la deuxième et la troisième perte, quant à elles, n'échappent pas au recours au timeout pour les détecter, à l'inverse de New-Reno (modifié).

5.5 Perspectives

Il est facile de vérifier, que lorsqu'une retransmission est perdue sous New-Reno (et toutes les autres versions), le recours au timeout est inévitable pour la recouvrir. On pourrait envisager d'utiliser les acquittements des paquets envoyés durant la phase de "fast recovery" pour y palier. Ainsi, si les paquets transmis durant le "fast

recovery" sont bien reçus par la destination, ou du moins avec de très faibles pertes (ex. un paquet sur 4), on pourrait recouvrir de toutes les pertes uniquement avec des acquittements dupliqués. Cela suppose bien évidemment que la destination envoie un acquittement pour chaque paquet dès sa réception.

Le mécanisme de "fast recovery" permet de transmettre de nouveaux paquets si la taille actuelle de la fenêtre le permet, en la faisant croître d'un paquet à chaque réception d'un acquittement dupliqué. Cependant, pour qu'on puisse détecter les pertes ayant lieu pour une deuxième fois, on devrait avoir envoyé des paquets durant cette phase ("fast recovery"). Du moment que la première perte est détectée par la réception de trois acquittements dupliqués, il est donc tout à fait raisonnable de transmettre au moins deux paquets en "fast recovery" pour remplacer les paquets ayant causé la réception de ces acquittements dupliqués. Dans ce qui suit, on propose d'envoyer au minimum trois paquets dans la phase de "fast recovery" en plus des paquets susceptibles d'être transmis durant son exécution "normale".

La proposition est la suivante :

Lors de la réception de trois acquittements dupliqués indiquant le commencement d'une phase de "fast retransmit", on compte le nombre de segments déjà transmis est non encore acquittés, qui est une indication sur le nombre maximal d'acquittements dupliqués qu'on pourrait recevoir après la première retransmission. Soit $Nack\_ancien = Snd\_nxt - Snd\_una - 1$ ce nombre. A la première retransmission, on envoie au minimum trois paquets en "fast racovery", et on compte le nombre de segment transmis durant cette phase, qu'on désigne par $Nack\_nouveau$. On compte également le nombre d'acquittements reçu par la source $t\_dupacks$. Si ce dernier est supérieur à $Nack\_ancien + 2$, alors on décide de retransmettre le premier paquet non encore acquitté. Cela signifie que les paquets envoyés en "fats recovery" continuent d'acquitter le dernier paquet bien reçu, et ça ne peut traduire que la perte du segment retransmis.

A la deuxième retransmission du même paquets, on envoie également au minimum trois paquets en phase de "fast recovery", on met à jour $Nack\_ancien = Nack\_ancien + Nack\_nouveau - 1$, et $Nack\_nouveau = 0$ tout en recommençant à compter le nombre de paquets transmis durant la phase de "fast retransmit" (y compris les retransmissions), on réitère alors la même procédure que précédemment. Tous les paramètres sont remis à zéro une fois qu'un segment plus grand que la plus grande séquence, envoyée juste avant le "fast retransmit", soit acquitté (à la fin de la phase de "fast retransmit").

On peut également réduire la fenêtre encore une fois, tout comme dans un “fast retransmit”, à la détection d’une perte qui a lieu pour une seconde ou troisième fois. Cela se justifie par le fait que les pertes successives d’un paquet peuvent signifier une très forte congestion dans le réseau. Cela risque cependant de diminuer le nombre de fois qu’une perte puisse être détectée. Les trois paquets transmis durant cette phase, peuvent venir à ce moment pour compenser la réduction de la fenêtre. Le diagramme 13 décrit en détails les instructions à exécuter pour recouvrir d’une perte ayant lieu une seconde fois.

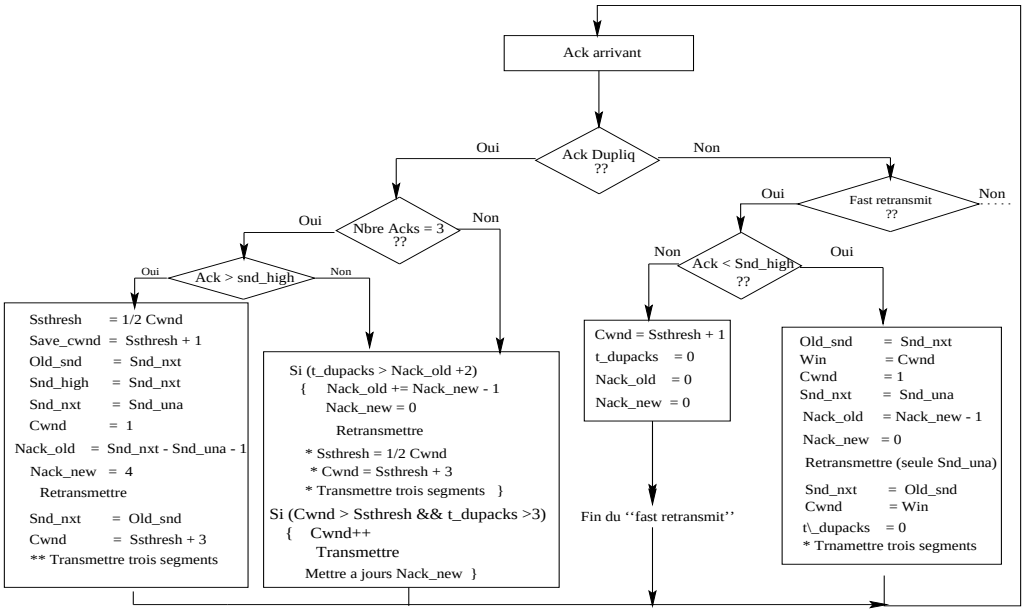


FIG. 13: Diagramme pour détecter la perte d’une retransmission

Sur Internet, le nombre de RTT nécessaires pour détecter une perte qui a lieu pour la deuxième fois, dépend de la probabilité de perte des paquets transmis en phase de “fast recovery”. Toutefois, il est tout à fait intéressant d’envisager de telles méthodes dans les réseaux à haut débit. Dans ces réseaux, le RTT est, généralement, moins d’un dixième de la granularité du timeout. Il est également très connu que le problème de TCP sur ATM réside essentiellement dans la granularité du timeout (500 ms) [27]. Si l’on ajoute à cela le fait que la reception d’un seul acquittement

dupliqué signifie une perte dans un réseau ATM, il est alors tout à fait possible de détecter toutes les pertes rien qu'en utilisant les acquittements dupliqués. Cela économiserait un temps considérable et par la même occasion accroîtrait le débit d'une manière spectaculaire.

La figure 14 (a), illustre un exemple où une retransmission est perdue, le paquet 360 en l'occurrence, pour une deuxième fois. On peut voir qu'à l'aide de la méthode décrite dans cette section (Seq-Pertes-Suc), il est tout à fait possible de la détecter, pendant que New-Reno (Seq-New-Reno) doit attendre près de deux secondes pour pouvoir la retransmettre, soit près de 20 fois le délai d'aller retour. La figure 14 (b), quant à elle, illustre la séquence des acquitements en fonction du temps, pour les différentes versions analysées dans cet article (Tahoe, Reno, New-Reno (modifié) et New-reno avec détection de perte des retransmissions), quand une retransmission est perdue. On peut bien constater que la dernière version excède en débit toutes les autres version, à cause du fait que ces dernières perdent, près de deux secondes pour détecter la perte d'une retransmission (paquet 360). A rappeler que l'amélioration proposée dans cette section ne change rien quant aux performances de New-Reno, en l'absence de retransmissions perdues (la nouvelle version a le même débit que New-Reno (modifié) dans ce cas).

On peut envisager d'utiliser uniquement les paquets envoyés durant le fast recovery par New-Reno, sans en envoyer trois segments après chaque retransmission, comme on l'avait proposé (sans les instructions précédées d'"*" dans le diagramme 13). A ce moment, le nombre de fois qu'une perte puisse être détectée se réduit. En effet, dans certains cas l'envoi de paquets après chaque retransmission est essentiel pour recevoir d'autres acquittements, mais avec le risque de saturer le réseau et par conséquent de les perdre lors de fortes congestions. Toutefois, après la première retransmission ("fast retransmit" précédé d'un "**" dans le diagramme 13) il est tout à fait raisonnable de transmettre trois segments dans la phase de "fast recovery" pour remplacer les trois acquittements dupliqués reçus.

Dans l'exemple de la figure 14, le nombre maximal de segments envoyés à la réception d'un acquittement est limité à quatre paquets, ceci pour éviter les rafales (bursts) lorsqu'un segment beaucoup plus grand que la séquence retransmise est acquitté.

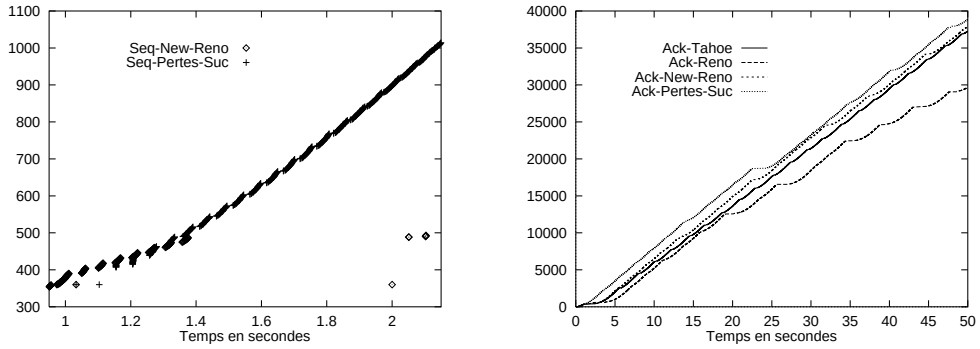


FIG. 14: Détection des retransmissions perdues, par acquittements dupliqués

6 Conclusion

Nous avons vu dans le present article toutes les versions de TCP ayant suscité un grand intérêt. Ainsi, nous avons gardé le lien chronologique et logique entre les différentes versions (Tahoe Reno New-Reno). Nous avons également commenté les nouvelles techniques apportées par Vegas. Cette dernière version, bien qu'elle permette de meilleures performances que Reno, consomme cependant beaucoup de temps dans l'estimation de la bande passante, qui à son tour repose sur l'estimation du plus petit RTT susceptible à des changements en fonction du trafic dans le réseau. Vegas voit ses performances se dégrader également quand le nombre de connexions actives devient important [2].

Nous avons illustré quelques problèmes par de simples exemples de simulation. Quelques uns de ses problèmes (faux "fast recovery" et faux "fast retransmit") ont fait l'objet d'une précédente publication [3]. Des problèmes similaires à ceux la ont été décrit dans [14], et une version de TCP, TCP New-Reno en l'occurrence a été proposée. Nous avons inclu les solutions au problèmes de "fast recovery" après timeout dans cette dernière, et discuté quelques détails techniques (problème des "fast retransmit" successifs dans New-Reno).

Enfin,nous avons proposé une méthode permettant de détecter des pertes sans avoir recours au timeout. Cette méthode peut sembler compliquée, mais ne nécessite pas un temps énorme d'exécution. Seules deux variables à rajouter suffisent. Son intérêt dans les réseaux de demain ne serait pas des moindres.

Références

[1] J.S. Ahn, P. B. Danzig, Z. Liu and L. Yan, "Experience with TCP Vegas : Emulation and experiment," *Proc. SIGCOMM'95 Symp.*, Aug. 1995.

[2] O. Ait-Hellal, E. Altman, "Analysis of TCP-Vegas and TCP-Reno", *IEEE International Conference on Communications (ICC'97)*, pp. 495-499. Montreal, 8-12 juin 1997. À paraître dans le journal de *Telecommunication systems*.

[3] O. Ait-Hellal, E. Altman, "Problems in TCP Vegas and TCP Reno", *DNAC (De Nouvelles Architectures pour les Communications)*, UVSQ, Paris, 3-5 décembre 1996.

[4] E. Altman, F. Boccara, J. Bolot, P. Nain, P. Brown, D. Collange and C. Fenzy "Analysis of TCP/IP Flow Control Mechanism in Hight- Speed Wide-Area Networks", *IEEE CDC Conference*, New Orleans, décembre 1995.

[5] O. Bonaventure, "A simulation study of TCP with the proposed GFR service category", Presented at DAGSTUHL seminar 9725, *High-Performance Networks for Multimedia Applications*, Schloβ Dagstuhl, Allemagne, 15-20 juin 1997..

[6] Lawrence Brakmo. "TCP Vegas Release 0.8", novembre 1994.

[7] L.S Brakmo, L.L. Peterson, "TCP Vegas : End to End Congestion Avoidance on a Global Internet," *IEEE JSAC*, vol. 13, pp. 1465-1480, 1995.

[8] D. D. Clark, "window and acknowledgement strategy in TCP", *RFC-813*, juillet 1982.

[9] D. E. Comer, J. C. Lin, "TCP Buffering And Performance Over An ATM Network", *Perdue Technical Report CSD-TR 94-026*, mars 1994.

[10] D. E. Comer, D. L. Stevens, "Internetworking with TCP/IP", *Volume 1, 2, 3. Addison Wesley*, 1994.

[11] J. Crowcroft, Ph. Oechslin, "Differentiated End to End Internet Services using a Weighted Proportional Fair Sharing TCP", draft submitted for publication.

[12] K. Fall, S. Floyd, "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP", *Proceedings of ACM SIGCOMM'95*, mai l996. disponible sur le site ftp à ftp.ee.lbl.gov/papers/.

[13] S. Floyd, "TCP and successive fast retransmits". *Lawrence Berkeley Laboratory, Technical Report*, février 24 1995.

[14] J. C. Hoe, “Improving the Start-up Behavior of a Congestion Control Scheme for TCP”, *In proceedings of the ACM SIGCOMM’96*, pages 270-280, septembre 1996.

[15] Van Jacobson. “Congestion avoidance and control”. *ACM SIGCOMM 88*, pages 273-288, 1988.

[16] Van Jacobson, “Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno”, *Proceedings of the Eighteenth Internet Engineering Task Force*, pp. 365, University of British Columbia, Vancouver, B.C, septembre 1990.

[17] Van Jacobson. “modified TCP congestion avoidance algorithm”, *mailling list, end2end-interest*, 30 avril 1990.

[18] Van Jacobson, R Braden, “TCP extensions for Long-Delay Paths”, *RFC 1072*, octobre 1988.

[19] Van Jacobson, R Braden, D. Borman, “TCP extensions for High Performance”, *RFC 1323*, mai 1992.

[20] Van Jacobson, L. Peterson, L. Brakmo, S. Floyd, “Problems with Arizona’s Vegas” *mailing list, end2end-tf* (discussion sur TCP Vegas), 1994.

[21] S. Keshav, “REAL : A network simulator”, *Comp. Sci. Dept., UC Berkeley, Technical Report* 88/472, 1988.

[22] T. V. Lakshman, U. Madhow “Performance analysis of window-based flow control using TCP/IP : the effect of high bandwidth-delay products and random loss”, *IFIP Transactions C-26, High Performance Networking*, pp. 135-150, North-Holland, 1994.

[23] T. V. Lakshman, U. Madhow, B. Suter “Window-based error recovery and flow control with a slow acknowledgment channel : a study of TCP/IP performance” submmis pour publication.

[24] V. Paxon, “Measurements and Analysis of End-to-End Internet Dynamics”, *Ph.D Thesis*, Computer Science Division, University of California, Berkley, avril 1997.

[25] K. Ramakrishnan, R. Jain, “A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with Connectionless Network Layer”, *Proc. SIGCOMM’88*, août 1988.

[26] A. Romanow, “TCP over ATM : Some Performance Results”, *ATM Forum/93-784*. ftp ://playground.sun.com/pub/tcp_atm/tcp_forum.7_93.ps.

[27] A. Romanow, S. Floyd, “The Dynamics of TCP Traffic over ATM Networks”, *ACM SIGCOMM Computer Communications Review*, octobre 1994.

[28] S. G. Sanjay, A. Kumar, “TCP Over End-to-End ABR : A Study of TCP Performance with End-to-End Rate Control and Stochastic Available Capacity”, *IEEE Globecom’98*, Sydney, 1998.

[29] W. R. Stevens. “TCP/IP illustrated”, *volume 1. Addison Wesley*, 1994.

[30] A. Tanenbaum, “Réseaux”, 3ème édition, *InterÉditions*, 1997.

[31] Z. Wang, J. Crowcroft, “A New Congestion Control Scheme : Slow Start and Search (Tri-S)”, *ACM Computer Communication Review*, 21(1), pp.32-43, janvier 1991.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399