TCP
Impact des Implémentations sur les Performances

Guillaume Melquiond Blaise Potard

16 mai 2002

Introduction

Le protocole TCP a été mis au point dans le cadre du projet de réseau Arpanet. En 1981 apparaît la première version standard dans le RFC 793 [1]. Elle s'appuie sur neuf éditions précédentes et clarifie le protocole.

Ce protocole est aujourd'hui celui qui est le plus utilisé au niveau mondial, puisqu'il est l'une des briques de base d'Internet. C'est un protocole connecté et fiable. Il permet de s'affranchir des problèmes de fragmentation, de réordonnancement et de perte de paquets.

Les principales recherches ultérieures sur ce protocole ont eu pour sujet le contrôle de flux. En effet, la première version n'implémentait pas de mécanisme de contrôle de congestion. Et ainsi, en 1988 apparaît la première implémentation standard de TCP avec de tels mécanismes : c'est la version Tahoe.

À partir de là, les différentes implémentations du protocole ont toutes cherché à améliorer ces différents mécanismes. D'autant plus que les types de connexions se sont diversifiés : les débits sont très variables et les liens peuvent ne pas être filaires. Il faut aussi pouvoir assurer l'équité des différentes connexions.

Parmi ces autres implémentations, il y a Reno (en 1990), Vegas (1994), SACK (1996), NewReno et NetReno (1998), Niagara (2001). Il existe de nombreuses autres variantes.

Chapitre 1

TCP

Les premières versions sont très anciennes (à l'échelle de l'informatique). Ici on s'intéresse à la première version "standard" – en réalité la neuvième version – définie dans le RFC 793[1]. Ce RFC décrit le format des paquets et les techniques à utiliser pour ordonner les paquets. Cependant, il ne spécifie pas les mécanismes de contrôle de flux, et en particulier il ne s'intéresse pas du tout aux problèmes de congestion.

1.1 Mécanismes de base de TCP

1.1.1 Transfert de données de base

TCP est capable de transférer un flux continu de données entre deux ordinateurs, en découpant ce flux en paquets ou datagrammes. En général, TCP décide de lui-même là où le flux de données doit être coupé.

1.1.2 Contrôle d'erreur

TCP doit considérer et traiter les cas de données perdues, erronées, dupliquées, ou arrivées dans le désordre à l'autre bout de la liaison Internet. Ceci est réalisé par l'insertion d'un numéro de séquence, et par l'obligation d'émission d'un "accusé de réception" (ACK) par le TCP destinataire. Si l'accusé de réception n'est pas reçu au bout d'un temps prédéfini, le paquet sera réémis. Côté récepteur, les numéros de séquence sont utilisés pour reconstituer dans le bon ordre le flux original, et éliminer les paquets dupliqués. L'élimination des erreurs physiques de transmission se fait par encodage d'un Checksum à l'émission, recalcul de ce Checksum par le destinataire, et élimination des paquets pour lesquels les deux valeurs ne correspondent pas.

Tant que TCP fonctionne correctement, et que le réseau Internet n'est pas saturé, aucune faute de transmission ne devrait transparaître dans la communication.

1.1.3 Multiplexage

Pour permettre à plusieurs tâches d'une même machine de communiquer simultanément via TCP, le protocole définit un ensemble d'adresses et de ports pour la machine. Une *socket* est définie par l'association des adresses Internet source, destinataire, ainsi que les numéros de port à chaque bout. Une connexion nécessite la mise en place de deux sockets. Une socket peut être utilisée par plusieurs connexions distinctes.

1.1.4 Connexions

Les mécanismes de fiabilisation et de contrôle de flux décrits ci-dessus imposent à TCP l'initialisation et la maintenance de certaines informations pour chaque communication. La combinaison de ces informations, dont les sockets, les fenêtres, et les numéros de séquence formeront ce que nous appelons une connexion. Chaque connexion est identifiée de manière unique par sa paire de sockets, définissant chacun des deux sens de la communication.

Lorsque deux processus désirent communiquer, les modules TCP doivent tout d'abord négocier et établir une connexion (initialiser ces informations d'état de part et d'autre). Une connexion peut servir à plusieurs échanges (simultanées ou consécutifs) ; pour pouvoir les différencier, on attribue à chacun de ces échanges un numéro de séquence initial, suffisament différents entre eux pour qu'il ne puisse pas y avoir de confusions dans l'attribution des paquets.

Dans la mesure où l'on considère que les ordinateurs, ainsi que le réseau Internet, ne sont pas d'une fiabilité absolue, on utilise une méthode d'initialisation par négociation bilatérale basée sur une horloge pour les numéros de séquence.

1.1.5 Contrôle de flux

TCP fournit un moyen au destinataire pour contrôler le débit de données envoyé par l'émetteur : en indiquant dans le champs “fenêtre” des ACK la quantité de données (en octets) qu'il est prêt à recevoir. Cette taille de fenêtre doit être bien choisie : si elle est trop petite, le transfert de données se fera lentement. Si elle est trop grande, le récepteur peut perdre des paquets, qui devront donc être renvoyés. En fait on s'aperçoit vite, sur des réseaux comme Internet, que ce qui pose problème n'est pas tellement la capacité de réception du destinataire, mais plutôt celles des machines situées entre elles (les noeuds du réseau).

1.2 Paquets

Les paquets TCP sont envoyés sous forme de datagrammes Internet. L'entête IP transmet un certain nombre de paramètres, tels que les adresses Internet source et destinataires. L'entête TCP est placée à la suite, contenant les informations spécifiques au protocole TCP. Cette division permet l'utilisation de protocoles autres que TCP, au-dessus de la couche IP. L'entête TCP est décrit dans la figure 1.1.

Port source : 16 bits
: Le numéro de port de la source.

Port Destinataire : 16 bits
: Le numéro de port du destinataire.

Numéro de séquence : 32 bits
: L'indice du premier octet de données par rapport au début de la transmission, sauf si SYN est marqué. Si SYN est marqué, le numéro de séquence est le numéro de séquence initial (ISN), le premier octet de données à pour indice ISN+1.

Accusé de réception : 32 bits
: Si ACK est marqué, ce champ contient le numéro de séquence du prochain octet que le récepteur s'attend à recevoir. Une fois la connexion établie, ce champ est toujours rempli.

Taille de l'entête : 4 bits

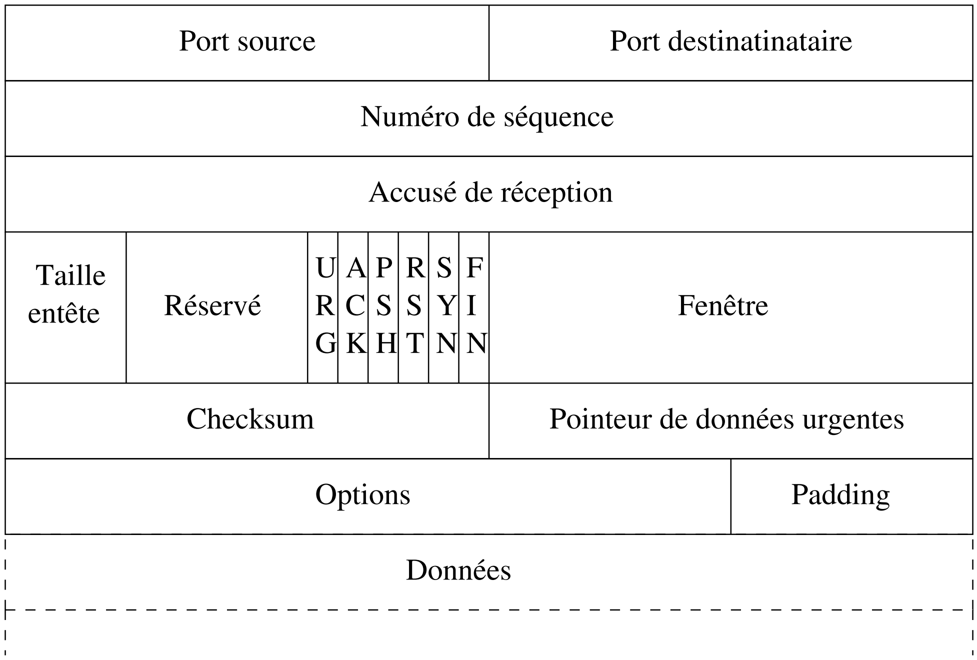


Fig. 1.1 – Entête TCP

La taille de l'entête TCP en nombre de mots de 32 bits. Il indique l'endroit où commence les données. L'entête TCP, dans tous les cas, a une taille correspondant à un nombre entier de mots de 32 bits.

Réservé : 6 bits

Réservés pour usage futur. Doivent nécessairement être à 0.

Bits de contrôle : 6 bits

- URG : Pointeur de données urgentes significatif
- ACK : Accusé de réception significatif
- PSH : Fonction Push
- RST : Réinitialisation de la connexion
- SYN : Synchronisation des numéros de séquence
- FIN : Fin de transmission

Fenêtre : 16 bits

Le nombre d'octets à partir de la position marquée dans l'accusé de réception que le récepteur est capable de recevoir.

Checksum : 16 bits

La somme de contrôle est constituée en calculant le complément à 1 sur 16 bits de la somme des compléments à 1 des octets de l'entête et des données pris deux par deux (mots de 16 bits). Si le message entier contient un nombre impair d'octets, un 0 est ajouté à la fin du message pour terminer le calcul. Cet octet supplémentaire n'est pas transmis. Lors du calcul de la somme de contrôle, les bits attribués à celle-ci dans l'entête sont marqués à 0.

Ce mécanisme couvre de plus une pseudo entête de 96 bits préfixée à l'entête TCP. Cette pseudo entête comporte les adresses Internet source et destinataires, le type de protocole et la longueur du message TCP. Ceci protège TCP contre les erreurs de routage. Cette information sera véhiculée par IP, et est donnée comme argument par l'interface TCP/Réseau lors des appels d'IP par TCP.

Pointeur de données urgentes : 16 bits

Communique la position d'une donnée urgente en donnant son décalage par rapport au numéro de séquence. Le pointeur doit pointer sur l'octet suivant la donnée urgente. Ce champ n'est interprété que lorsque URG est marqué.

Options : variable

Les champs d'option peuvent occuper un espace de taille variable à la fin de l'entête TCP. Les options définies ne sont pas très nombreuses, on peut essentiellement y définir la taille maximal d'un segment. C'est aussi ici que sont inscrites les données SACK.

Bourrage (padding) : variable

Les octets de bourrage terminent l'entête TCP de façon à l'aligner sur 32 bits et de sorte que la taille de l'entête indiquée soit correcte.

1.3 Transfert de données

Une fois la connexion établie, les données peuvent commencer à être échangées dans des segments. Ceux-ci pouvant être rejetés (erreur de Checksum), ou perdus dans le réseau (congestion du réseau), TCP utilise un principe de retransmission (au bout d'une temporisation) pour garantir l'acheminement de tous les segments de données. Ce principe de retransmission peut conduire à la réception de "doublons". TCP effectue un certain nombre de tests sur les séquences et les acquittements afin de déduire l'acceptabilité des données reçues.

1.3.1 Time-out

Du fait de la grande variété de réseaux formant Internet, et l'emploi massif de connexions TCP à travers ce réseau la valeur de temporisation de retransmission doit être déterminée dynamiquement. Pour cela, il est recommandé d'utiliser la méthode suivante : on mesure un RTT (*Round-Trip Time*), c'est à dire le temps écoulé entre l'envoi d'un segment et la réception d'un ACK englobant le numéro de séquence de ce segment. On corrige cette valeur en calculant le SRTT (*Smoothed RTT*) sous la forme d'une moyenne géométrique des RTT :

$$SRTT_{n+1} = \alpha * SRTT_n + (1 - \alpha) * RTT$$

Enfin, on calcule la valeur du RTO (*Retransmission Time-Out*) comme cela :

$$RTO = \min(T_{max}, \max(T_{min}, (\beta * SRTT)))$$

où T_{max} est une borne supérieure pour la temporisation (par exemple 1 minute), T_{min} une borne inférieure (par exemple 1 seconde), α le “facteur de propagation” (par exemple $0.8 \leq \alpha \leq 0.9$), et β un facteur de variance du temps de propagation (par exemple $1.3 \leq \beta \leq 2.0$).

1.3.2 Fenêtres de transmission

La fenêtre de transmission designe l'intervalle de données (dans la séquence totale transférée) qui peut être traité simultanément par l'une ou l'autre des personnes connectées. Par exemple, la fenêtre de transmission de l'émetteur indique les paquets pouvant être envoyés. Ces fenêtres sont

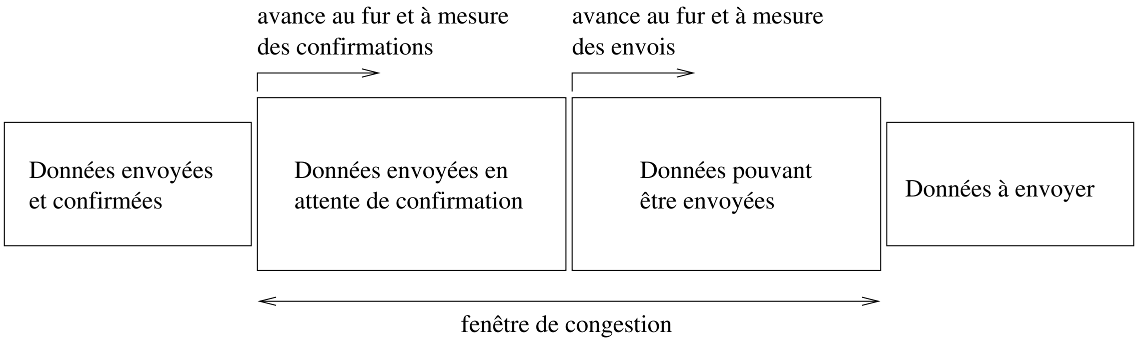


FIG. 1.2 – Fenêtres de l'émetteur

glissantes : lorsque les paquets du début de sa fenêtre arrivent chez le destinataire, sa fenêtre glisse vers la droite. De même quand l'émetteur reçoit des acquittements pour des paquets au début de sa fenêtre, sa fenêtre se déplace.

Une fenêtre de petite taille a une plus grande fiabilité, et le nombre de paquets réémis est réduit au minimum. Mais par contre, cela occupe très mal la bande passante. Une fenêtre de grande taille est beaucoup moins fiable, il y a beaucoup de risque de perdre des paquets, et si la fenêtre est trop grande, on court le risque de réémettre un grand nombre de fois les mêmes paquets. Par contre on utilise beaucoup mieux la bande passante disponible. En fait, l'objectif pour une implémentation de TCP est de trouver la taille de fenêtre "idéale" en un minimum de temps, et également être capable de s'adapter aux fluctuations du trafic. L'émetteur et le récepteur ont chacun une fenêtre de transmission, qui peuvent être de tailles différentes. Le récepteur indique dans ses "ACK" la taille de sa fenêtre de réception, ce qui permet à l'émetteur de savoir jusqu'où il peut aller.

Les fenêtres d'émission et de réception ne sont pas forcément de la même taille. Si la fenêtre de réception est plus petite que celle d'émission cela pose un problème, mais il se résout facilement puisque l'on a la taille de cette fenêtre dans les accusés de réception.

Par contre, dès que des réseaux de taille conséquente commencèrent à apparaître, le problème des congestions au niveau des noeuds internes du réseau se posa.

Chapitre 2

TCP Tahoe

Cette implémentation du protocole a été proposée par Van Jacobson en 1988 et a été introduite dans la version 4.3 de BSD. Son grand apport est constitué des algorithmes de contrôles de flux pour éviter les congestions du réseau et accélérer les transmissions. Ces algorithmes sont décrits dans [3].

2.1 Algorithmes

La taille de la fenêtre de réception (notée *rwnd*) limite le nombre de paquets que l'émetteur peut envoyer sans recevoir de confirmation, cela permet d'éviter de saturer le récepteur. La taille de la fenêtre de congestion de l'émetteur (notée *cwnd*) limite aussi ce nombre de paquets, l'objectif étant cette fois de ne pas saturer le réseau.

Au départ, la fenêtre de congestion a la taille d'un paquet et la variable *sstresh* est initialisée à la taille de la fenêtre de réception.

Pour les différentes figures, la taille de la fenêtre de réception est fixée à dix paquets. Et c'est aussi la valeur initiale de *sstresh*. Sous forme de carrés rouges sont indiqués les paquets émis. Les triangles verts indiquent le moment où le ACK correspondant est reçu par l'émetteur. Le temps est représenté horizontalement, tandis que les numéros de paquets sont représentés verticalement (modulo 60).

2.1.1 Slow start

Le démarrage lent est contrôlé par un seuil noté *sstresh* (pour *slow start threshold*). Tant que la fenêtre de congestion reste plus petite que ce seuil, alors elle est augmentée de la taille d'un paquet à chaque réception d'un ACK.

Le démarrage est donc vraiment lent puisqu'un seul paquet est envoyé au départ, mais à la réception de sa confirmation d'arrivée, deux paquets peuvent être envoyés. Une fois ces paquets confirmés, quatre pourront être envoyés. Et ainsi de suite, le nombre de paquets émis double à chaque fois comme le montre la figure 2.1, ce qui assure d'atteindre rapidement un débit conséquent, sans pour autant avoir saturé le réseau par l'envoi brusque d'un grand nombre de paquets.

2.1.2 Congestion avoidance

Une fois que la taille de la fenêtre de congestion a dépassé le seuil de démarrage lent, le protocole passe dans ce mode. Cette fois, *cwnd* n'est plus augmenté de 1 paquets à chaque réception d'un ACK, mais seulement après chaque écoulement d'un délai RTT.

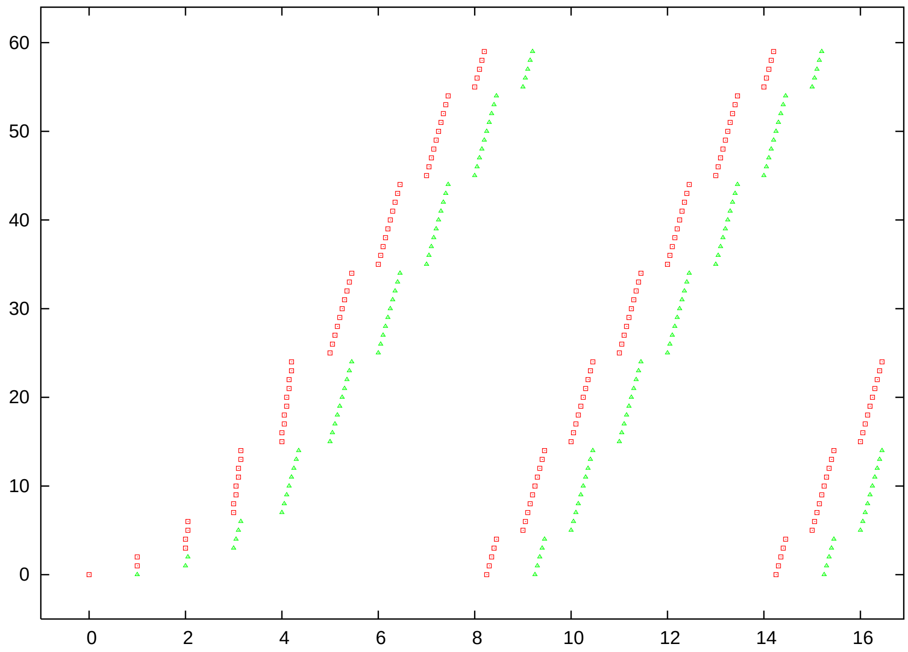


FIG. 2.1 – Algorithme *slow start*

En pratique, pour éviter de maintenir une variable supplémentaire, la fenêtre de congestion reste incrémentée à chaque réception d'un ACK, mais de la valeur suivante : $MSS * MSS/cwnd$. Comme, en régime permanent, la taille de la fenêtre de congestion correspond à la quantité de données envoyées pendant un RTT, l'émetteur reçoit $cwnd/MSS$ messages de confirmation, ce qui justifie la formule précédente.

Les figures 2.2 (à partir du temps 10) et 3.1 (à partir du temps 8) sont des exemples de *congestion avoidance* puisque *sstresh* vaut dans les deux cas 5 paquets, alors que la taille limite de la fenêtre de congestion est de 10 paquets.

2.1.3 Fast retransmit

Cependant, à moins que l'émetteur n'atteigne le débit maximal sur son lien en sortie ou que *cwnd* se retrouve bloqué à *rwnd*, le nombre de paquets contenus dans la fenêtre de congestion va augmenter jusqu'à ce qu'une congestion du réseau se produise. La conséquence est immédiate, un ou plusieurs paquets sont perdus.

Contrairement au protocole standard qui se contente d'envoyer un ACK pour chaque paquet dont les précédents sont arrivés, le protocole Tahoe envoie un ACK pour chaque paquet. Mais cela reste cohérent avec le protocole standard puisque ce sont des ACK pour le dernier paquet dont les précédents étaient arrivés.

Le protocole passe dans le mode de retransmission rapide quand il reçoit un ACK dupliqué trois fois. Cela signifie que le récepteur a reçu trois paquets plus récents que celui qu'il espérait obtenir. Le paquet en question est alors considéré par l'émetteur comme étant perdu et il est retransmis. Trois ACK dupliqués sont attendus (ce qui fait, en tout, quatre ACK pour un même paquet) afin d'éviter que ce mécanisme de retransmission rapide soit déclenché par un simple réordonnancement des paquets à la réception.

Quand l'émetteur passe en mode *fast retransmit*, la valeur de *ssthresh* est alors fixée à la moitié du nombre de paquets en cours de transfert à ce moment là (c'est-à-dire $\min(cwnd, rwnd)/2$ et *cwnd* reprend sa valeur initiale de 1 paquet. Par conséquent, le protocole repasse à nouveau dans le mode *slow start*.

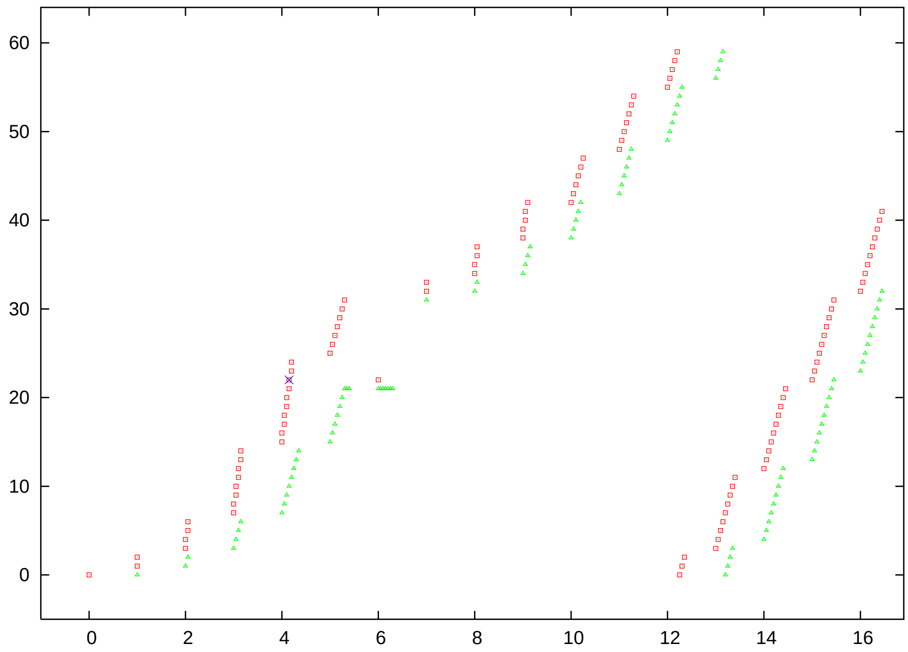


FIG. 2.2 – Algorithme *fast retransmit*

Cependant, malgré le fait que ce soit un démarrage lent, le terme de *fast retransmit* est justifié puisque l'émetteur n'a pas à attendre l'expiration d'un temporisateur avant de renvoyer le paquet perdu. Sur la figure 2.2 (inspirée de [7]), le paquet perdu est renvoyé au temps 6, sans avoir à attendre l'expiration d'un temps mort. Et dès le temps 7, l'envoi des paquets reprend depuis l'endroit où il s'était arrêté. Au temps 10, le protocole passe en phase de *congestion avoidance*.

2.1.4 Delayed ACK

Le principe est cette fois de ne pas confirmer chaque paquet reçu, mais seulement un sur deux. L'avantage de cette méthode est clair : si les transferts n'ont lieu que dans un sens, c'est-à-dire si le récepteur n'envoie que des paquets de confirmation, le trafic du récepteur vers l'émetteur est divisé par 2. Cela peut aussi permettre d'attendre que le récepteur ait des données à envoyer. Cependant, si jamais il y a un trop long délai et si un ACK est en suspens, il est envoyé, afin de ne pas bloquer l'émetteur.

Pour que cet algorithme puisse s'insérer correctement dans le reste du dispositif, il faut qu'il soit uniquement utilisé lorsque tout se passe bien. En effet, l'émetteur a parfois besoin de recevoir un nombre de ACK bien défini. Par exemple, en cas de réception de paquets dans le désordre, il faut que le récepteur se remette à envoyer des ACK pour chaque paquet reçu, afin que l'émetteur puisse passer en mode *fast retransmit* si nécessaire. Il risquerait sinon de ne jamais utiliser cet algorithme, puisqu'il faudrait qu'il reçoive l'équivalent de six anciens ACK dupliqués.

2.2 Performances

Le protocole initial n'implémentait pas vraiment de mécanismes de contrôle de flux, il n'était donc pas dur d'améliorer TCP sur ce point. En effet, les premières versions avaient tendance à envoyer d'un seul coup toutes les données que le récepteur pouvaient recevoir, ce qui pouvait facilement provoquer des congestions. D'autre part, elles attendaient l'expiration d'un long délai avant de réémettre les données qui s'étaient perdues, ce qui n'était pas un phénomène rare au vu du premier point.

Chapitre 3

TCP Reno

Cette version du protocole date de 1990 et a elle aussi été implémentée dans la version 4.3 de BSD. Elle est actuellement considérée comme le standard sur Internet. Elle n'apporte qu'un changement à la version Tahoe, c'est l'ajout de l'algorithme de *fast recovery*. Cet algorithme est décrit dans [3].

3.1 Fast recovery

Cet algorithme remplace en réalité l'algorithme de *fast retransmit*. Une fois les trois duplicata reçus, le paquet supposé manquant est renvoyé de la même façon que dans l'algorithme du protocole Tahoe. Mais ensuite, le comportement devient différent, l'émetteur ne passe plus en mode *slow start* mais en mode *fast recovery*.

En effet, comme un ACK dupliqué n'est envoyé qu'à chaque retrait d'un paquet par le récepteur, cela signifie qu'au moins trois paquets ont été retirés. L'algorithme de *fast recovery* cherche à refléter cette constatation. Tout comme précédemment, le seuil *ssthresh* se voit assigné comme valeur la moitié du nombre de paquets en cours de transfert. Quant à la variable *cwnd*, elle ne reprend pas sa valeur initiale, mais la valeur *ssthresh* plus le nombre de ACK dupliqués reçus.

Dès que la nouvelle valeur de *cwnd* le permet, de nouveaux paquets sont envoyés. Quand le paquet qui a été supposé perdu lors de l'entrée dans le mode *fast recovery* a été confirmé par le récepteur, l'émetteur sort de ce mode et fixe la valeur de *cwnd* à *ssthresh*.

La figure 3.1 montre la perte d'un paquet, suivi par l'entrée en mode *fast recovery* à la réception du troisième ACK dupliqué. La valeur de *sstresh* est alors fixée à 5 paquets. Aucun nouveau paquet n'est envoyé durant la période de récupération, parce qu'interdit par la taille de la fenêtre de réception. Et une fois tous les paquets confirmés, *cwnd* est réinitialisé à la valeur de *sstresh*. Comme la fenêtre de congestion est alors vide, 5 paquets sont envoyés à la fois, puis l'émetteur passe en mode *congestion avoidance* est augmente progressivement le nombre de paquets envoyés à chaque fois, jusqu'à atteindre de nouveau 10.

3.2 Performances

Comme le montrent les figures 2.2 et 3.1, l'implémentation Reno du protocole récupère bien plus vite de la perte d'un paquet puisqu'au temps 17, 14 paquets supplémentaires sont partis.

Cependant, cette implémentation Reno a beaucoup de mal à se remettre de la perte de plusieurs paquets. Ainsi, les figures 3.2 et 3.3 sont un exemple de perte de 2 paquets consécutifs sous Tahoe

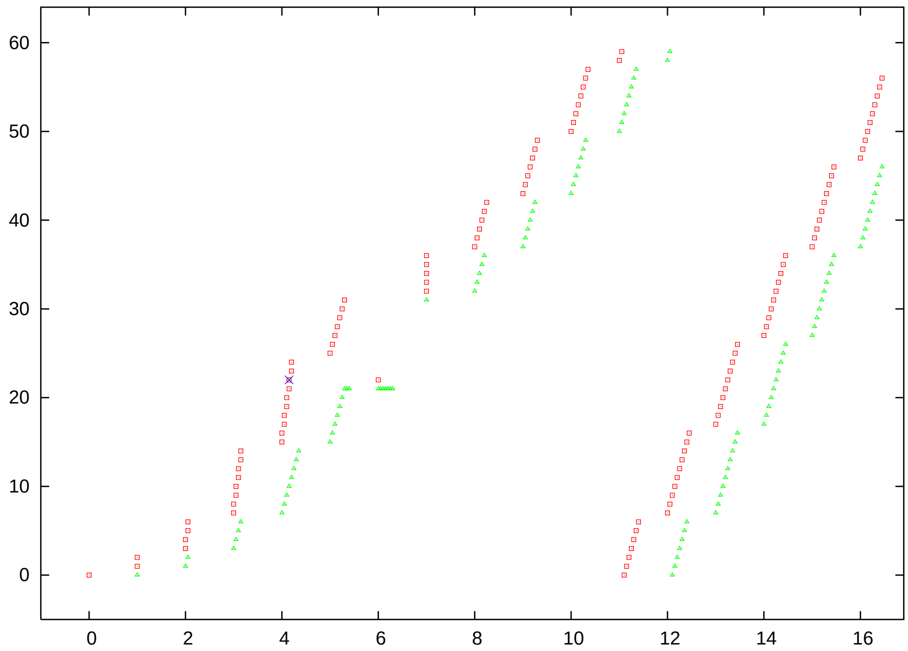


FIG. 3.1 – Algorithme *fast recovery*

et Reno.

Dans cet exemple, Tahoe va redémarrer normalement, même si un paquet se retrouve envoyé deux fois. Quant à Reno, le ACK reçu pour le premier paquet perdu provoque la sortie du mode *fast recovery*. Mais l'émetteur est incapable d'envoyer le paquet suivant parce que sa fenêtre de congestion est alors trop petite. Il est par conséquent obligé d'attendre que le délai de retransmission soit écoulé et qu'une erreur soit déclenché.

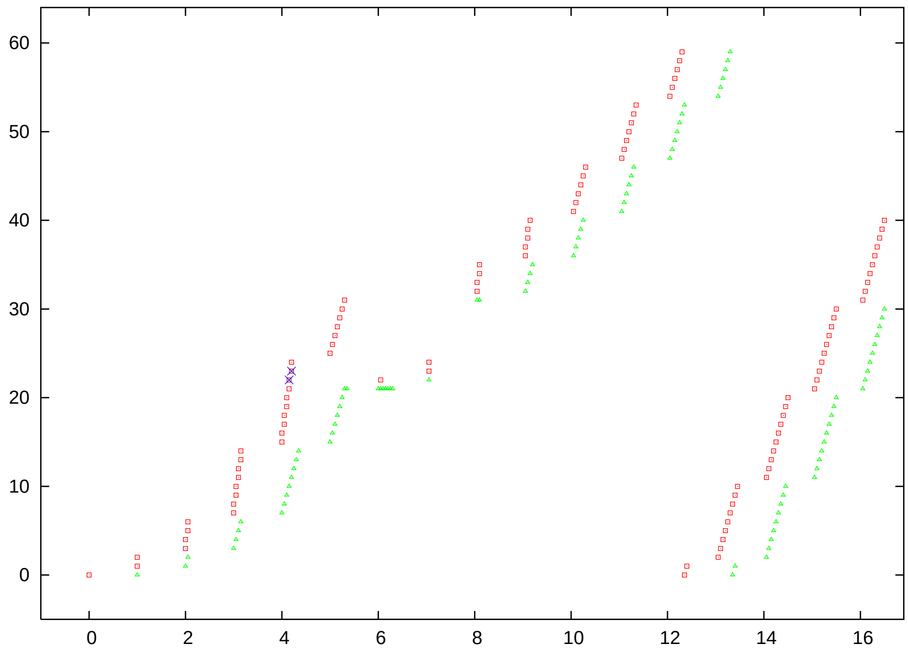


FIG. 3.2 – Perte de deux paquets sous Tahoe

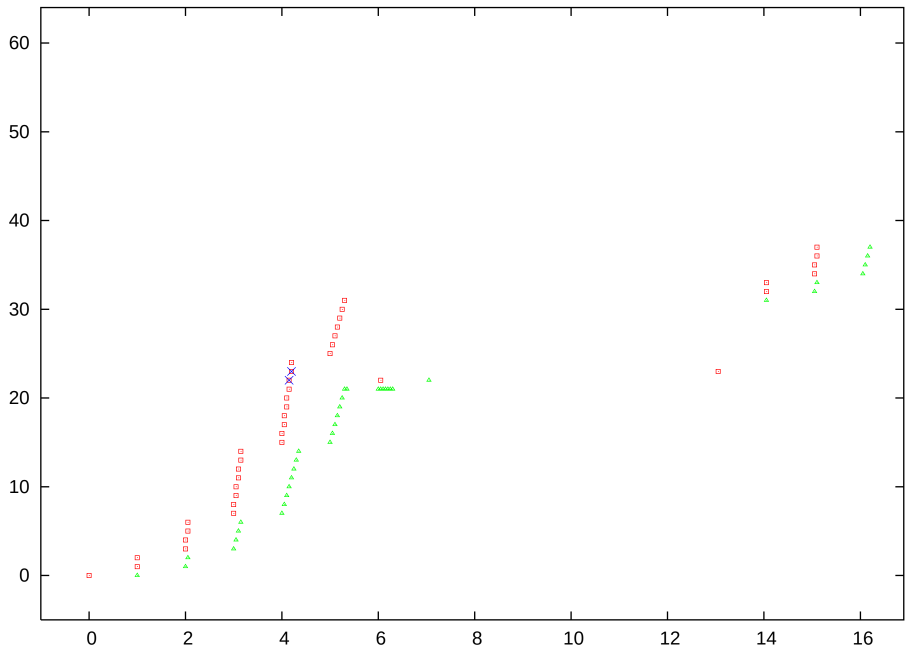


FIG. 3.3 – Perte de deux paquets sous Reno

Chapitre 4

TCP Vegas

4.1 Présentation

Vegas (décrite dans [6]) est une implémentation de TCP assez récente (elle date de 1994), et efficace. Elle donne en effet de bien meilleurs résultats que Reno, en envoyant les données plus vite, tout en en perdant moins (ses performances ne sont donc pas réalisées au détriment des autres connexions). Vegas utilise essentiellement trois techniques pour augmenter le débit et diminuer les pertes : un nouveau mécanisme de retransmission des paquets perdus, un mécanisme d'anticipation de congestion, et un mécanisme de *slow start* modifié pour limiter les pertes de paquets à ce niveau.

4.2 Nouveau mécanisme de retransmission

Dans Reno, le temps d'un aller-retour est calculé en utilisant un timer à "gros grain" (à peu près une demi-seconde), ce qui fait que les estimations ne sont pas très précises. Cela a des conséquences sur les temps eux-mêmes, mais aussi sur l'intervalle de temps à attendre avant un time-out.

Vegas étend les mécanismes de retransmission de Reno de la façon suivante. Pour chaque segment envoyé, Vegas lit et enregistre l'horloge système. Quand un ACK arrive, Vegas lit à nouveau l'horloge et calcule le temps de l'aller-retour (RTT) avec cette valeur et la valeur initiale. Vegas utilise cette estimation plus précise du RTT pour décider de retransmettre dans les cas suivants :

- Quand un ACK est reçu deux fois, Vegas regarde si la différence entre le temps courant et la valeur enregistrée pour le segment "candidat" est plus grande que la valeur du time-out. Si c'est le cas, Vegas retransmet le segment sans attendre d'avoir reçu le ACK en 3 exemplaires. En effet, dans beaucoup de cas, on peut perdre trop de paquets, ou la fenêtre être trop petite, pour que l'on reçoive 3 ACK identiques.
- Quand un ACK (que l'on n'a pas reçu avant) est le premier ou le deuxième après une retransmission, Vegas vérifie que le segment voulu n'est pas en time-out. Si c'est le cas, alors on retransmet le segment. Cela permet de rattraper les pertes de paquets multiples (en effet, quand on en arrive au point de perdre un paquet, il est tout à fait probable d'en perdre un autre).

En résumé, Vegas traite la réception de certains paquets comme "déclencheur" pour vérifier que l'on n'a pas de time-out sur certains paquets. Il utilise en parallèle le mécanisme de détection de time-out "à gros grains" de Reno, au cas où ces algorithmes ne détecteraient pas certaines pertes de paquets.

Ces mécanismes permettent à Vegas de détecter les pertes de paquets beaucoup plus rapidement que pour Reno, ce qui permet également d'être plus souple dans la gestion de la fenêtre de congestion. Vegas ne diminuera la fenêtre de congestion que si le segment retransmis a auparavant été envoyé *après* la dernière diminution (ce qui n'était pas forcément le cas pour Reno).

4.3 Anticipation de congestion

Reno a vis-à-vis des congestion une approche réactive : il ne diminuera la taille de la fenêtre de la congestion que lorqu'il aura perdu des paquets, c'est-à-dire quand il y aura effectivement eu une congestion. Vegas a une approche proactive et essaie de détecter les congestions avant qu'elles n'aient eu lieu, ce qui n'est pas si difficile. En effet, lorque la connexion commence à être encombrée, les files d'attentes au niveau de routeurs s'allongeant, le temps de transmission d'un paquet (le RTT) augmente. Vegas utilise cette idée pour éviter les congestions. L'algorithme utilisé effectivement est le suivant :

- on commence par calculer une valeur de RTT, appelée $BaseRTT$, que l'on considère être celle d'un paquet quand la connexion n'est pas encombrée, c'est typiquement le RTT obtenu pour le premier segment envoyé par la connexion. En pratique, dans Vegas, on utilise le minimum de tous les RTT mesurés. Si l'on ne sature pas la connexion, alors le débit attendu vérifie la relation : $Attendu = cwnd/BaseRTT$, où $cwnd$ désigne la taille (en octets) de la fenêtre de congestion,
- ensuite, on calcule le débit réel : on considère un segment particulier, on regarde combien d'octets sont transmis entre le moment où on commence à l'envoyer et le moment où on reçoit son ACK, et on divise ce nombre par le RTT observé pour le segment,
- enfin, on compare le débit réel au débit attendu, et on ajuste la taille de la fenêtre en conséquence. Soit $Diff$ la différence entre le débit attendu et le débit réel. Par définition, $Diff$ est positive. On fixe deux seuils α et β ($\alpha < \beta$). Si $Diff < \alpha$, alors on considère que l'on peut encore augmenter la fenêtre, et on l'augmente (linéairement). Si $Diff > \beta$, alors on considère que la fenêtre est trop grande et on la diminue (linéairement). On ne change pas la fenêtre de congestion quand $\alpha < Diff < \beta$.

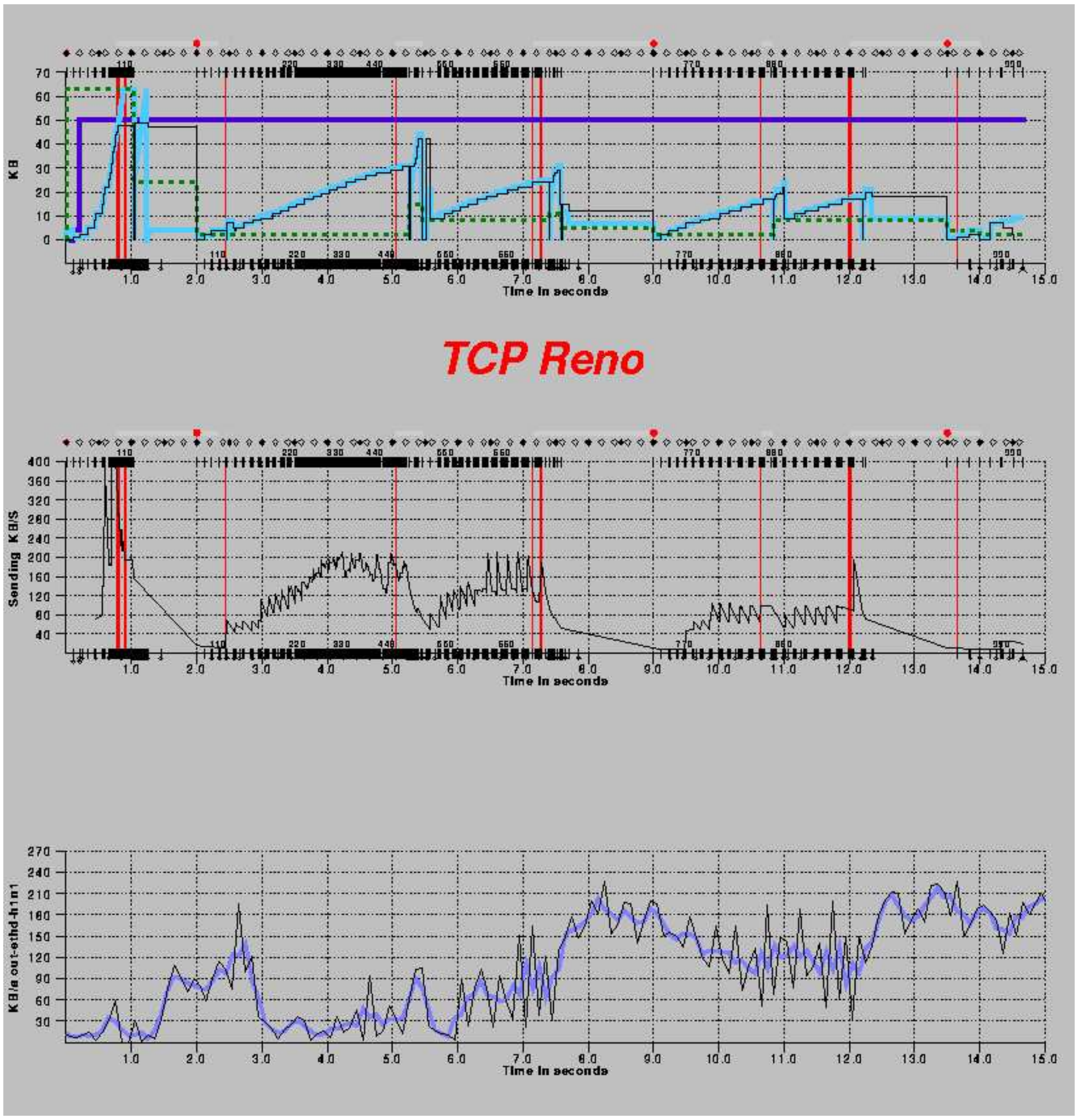
4.4 Mécanisme de slow start

L'algorithme de *slow start* de Reno coûte très cher en terme de paquets perdus (quand il n'est pas limité par un buffer d'émission trop petit ou une machine trop lente. Comme la fenêtre de congestion double de taille tous les RTT tant qu'il n'y a pas de perte, on peut s'attendre, quand il y a enfin saturation de la connexion à ce que la moitié des paquets de la fenêtre de congestion soient perdus. Au fur et à mesure de l'augmentation de la bande passante des réseaux, on peut s'attendre à perdre un nombre de plus en plus important de paquets, ce qui est un comble !

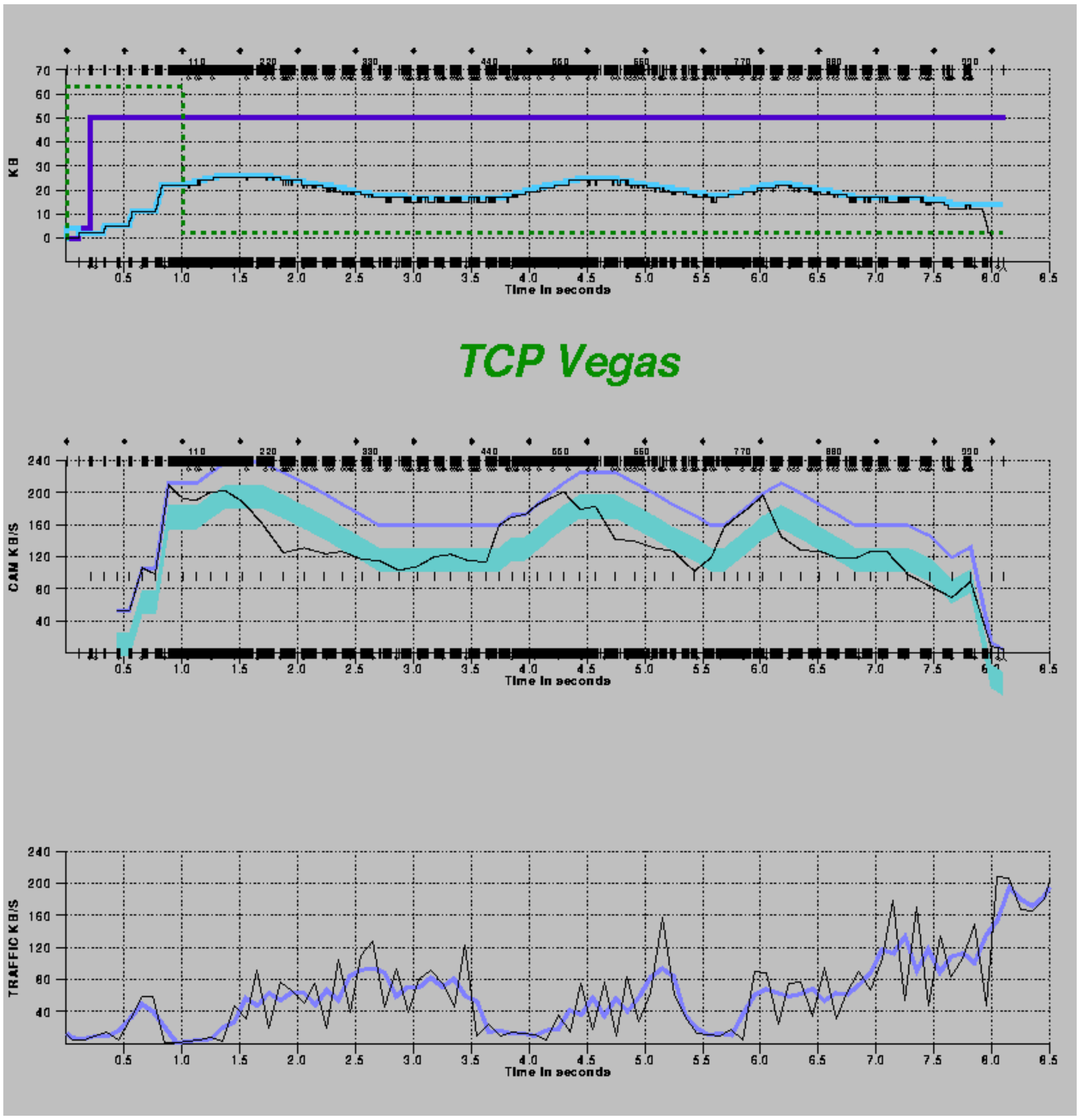
Pour corriger ce défaut, Vegas incorpore à l'algorithme de *slow start* son mécanisme de détection de congestion (légèrement modifié). Pour pouvoir détecter effectivement les congestions, on ne double plus la taille de la fenêtre tous les RTT, mais la taille reste fixée pendant quelques RTT pour qu'une comparaison valide puisse être fait entre le débit attendu et le débit réel. Quand la différence entre le débit réel et le débit attendu dépasse un certain seuil γ, on passe dans le mode de fonctionnement "normal".

4.5 Comparaison de Vegas et Reno

Pour comparer les performances de Vegas et Reno, on réalise une simulation où le protocole transfère un fichier de 1 Mo. Sur la figure 4.5, on a représenté les résultats pour TCP Reno. Le graphe du haut représente divers paramètres de contrôle. La courbe bleu clair représente la taille de la fenêtre de congestion. Le graphe du milieu représente le débit moyen (mesuré tous les 12 paquets). Les traits rouge verticaux dans ces deux graphes représentent les paquets perdus. Le graphe du bas représente le débit moyen du trafic de fond pendant le test.



La figure 4.5 représente cette fois les résulats pour TCP Vegas. Les graphes du haut et du bas représentent la même chose que précédemment, celui du milieu représente en noir le débit mesuré, et l'intervalle attendu (courbe bleue très épaisse). La taille de la fenêtre de congestion diminue quand le débit observé est en-dessous de l'intervalle attendu, augmente quand il est au-dessus, et ne change pas quand il est dans l'intervalle. On observe que TCP Vegas n'a perdu aucun paquet, et que le temps de transmission est beaucoup plus court que pour TCP Reno : on avait mis un peu moins de 15 secondes avec Reno, et on met un peu plus de 8 secondes avec Vegas.



Le tableau suivant récapitule les résultats obtenus par les concepteurs de l'algorithme sur Internet, les tests ont été réalisés entre l'Université d'Arizona et le National Institutes of Health. La connexion est constituée de 17 routeurs et passe par Denver, St. Louis, Chicago, Cleveland, New York et Washington. Ces résulats ont été obtenus en faisant la moyenne sur 49 envois (à diverses heures) d'un fichier de 1 Mo. Vegas donnait de meilleurs résultats que Reno dans 92% des tests. Les indices (respectivement 1,3 et 2,4) après Vegas indiquent les valeurs utilisées (en nombre de paquets) pour les seuils α et β.

	Reno	Vegas-1,3	Vegas-2,4
Débit (Ko/s)	53.0	72.5	75.3
Débit	100	137	142
Retransmissions (Ko)	47.8	24.5	29.3
Retransmissions	100	51	61
Time-outs (gros grain)	3.3	0.8	0.9

On peut observer également que sur une connexion Internet le mécanisme de détection de congestion ne fonctionne pas très bien (puisque Vegas perd tout de même une grande quantité de paquets), mais Vegas obtient tout de même de bien meilleurs résultats que Reno.

Il reste un autre point très important (qui explique sans doute pourquoi personne n'utilise Vegas aujourd'hui malgré ses excellentes performances) : lorsqu'une connexion Vegas est mise en concurence avec un protocole plus aggressif (Tahoe, Reno, NewReno, SACK...), la connexion Vegas se retrouvera systématiquement désavantagée en terme de bande passante si les machines ont des tampons de taille importante (car dans ce cas le mécanisme d'anticipation de congestion limitera systématiquement la quantité de données envoyée). Pour un tampon de taille 30 par exemple, Vegas obtient une bande passante au moins deux fois fois plus faible que celle de l'autre connexion.

Chapitre 5

TCP NewReno

5.1 Présentation

NewReno est une implémentation récente (1998) de TCP – définit dans le RFC 2582 [4] – qui améliore Reno. Elle en diffère assez peu : seuls les algorithmes de *fast retransmit* et *fast recovery* sont modifiés. On appelle *flight size* la quantité de données envoyées pour lesquelles on n'a pas encore reçu de ACK. L'algorithme de *fast retransmit* devient :

1. Quand le troisième ACK dupliqué est reçu, et que l'émetteur n'est pas encore dans la procédure de *fast recovery*, on prend pour *sstresh* la valeur suivante : $sstresh = \max(Flightsize/2, 2 \times MSS)$. On enregistre l'indice transmis le plus grand (parmi ces ACK) dans la variable *recover*.
2. On réémet le paquet perdu et on fixe la taille de la fenêtre de congestion à $sstresh + 3 \times MSS$. On agrandit artificiellement (par l'ajout de $3 \times MSS$) la taille de la fenêtre de congestion pour refléter le nombre de segments que le récepteur a reçus et enregistrés.
3. Pour tous les ACK dupliqués que l'on reçoit, on augmente d'un MSS la taille de la fenêtre de congestion.
4. On envoie un segment, si la nouvelle taille de la fenêtre de congestion et la fenêtre de réception du destinataire le permettent.
5. Quand un ACK pour de nouvelles données arrive :
   - Si ce ACK englobe toutes les données jusqu'à *recover*, alors c'est que tous les segments intermédiaires envoyés entre l'émission originale du paquet perdu et le troisième ACK ont été reçus. Dans ce cas, on donne comme nouvelle valeur à la taille de la fenêtre de congestion soit $\min(sstresh, Flightsize + MSS)$ soit *sstresh*. Remarquez que la valeur de *Flightsize* n'est plus la même qu'à la première étape. Si on choisit la seconde option, il faut prendre des précautions pour éviter des pic de données, si jamais la quantité de données en transit dans le réseau était beaucoup plus faible que ce que la nouvelle fenêtre de congestion permet. Puis on sort de la procédure de *fast recovery*.
   - Si ce ACK n'englobe pas toutes les données jusqu'à *recover*, alors c'est un "ACK partiel", et dans ce cas : on retransmet le premier segment non acquitté, on diminue la fenêtre de congestion de la quantité de données acquittée, on ajoute un MSS, puis on envoie un nouveau segment si la nouvelle taille le permet. Cette diminution de la taille de la fenêtre sert à faire en sorte que l'on ait une quantité de données "en transit" approximativement égale à *sstresh* quand on sortira de la procédure de *fast recovery*. On n'en sort pas pour l'instant (i.e., si un ACK dupliqué arrive, on exécute les étapes 3 et 4). On remet le timer de retransmission à 0 (si jamais ce timer dépasse un certain seuil, on repart en *slow start*).

5.2 Comparaison de Vegas et NewReno

Il y a assez peu de comparaison de Vegas et de NewReno, car Vegas n'a jamais été vraiment utilisé en pratique. On ne comprend pas vraiment pourquoi personne n'a voulu utiliser Vegas, peut-être parce que, mise en concurrence avec des connexions utilisant des protocoles aggressifs comme Reno, une connexion Vegas n'arrive pas à prendre une bande passante aussi importante que les autres ? Aujourd'hui, Reno est encore un protocole très utilisé ; mais NewReno est tout de même le plus utilisé. NewReno semble cependant obtenir des performances plus faibles que celles de Vegas dans la plupart des cas. Sur un réseau (vide) à 10 Mb, des tests de performance bruts donnent à NewReno une performance de 8.59 Mb, alors que Vegas arrive à obtenir 9.59 Mb.

5.3 Comparaison de Reno et NewReno

NewReno ne présente pas de grande différence avec Reno, mais il corrige (en partie) son principal défaut (qui est de perdre énormément de temps lorsque plusieurs paquets sont perdus). Les figures 3.3 et 5.1 illustrent bien ce phénomène. Mais il ne le corrige pas entièrement. La figure 6.1 montre comment NewReno réagit quand on perd 3 paquets.

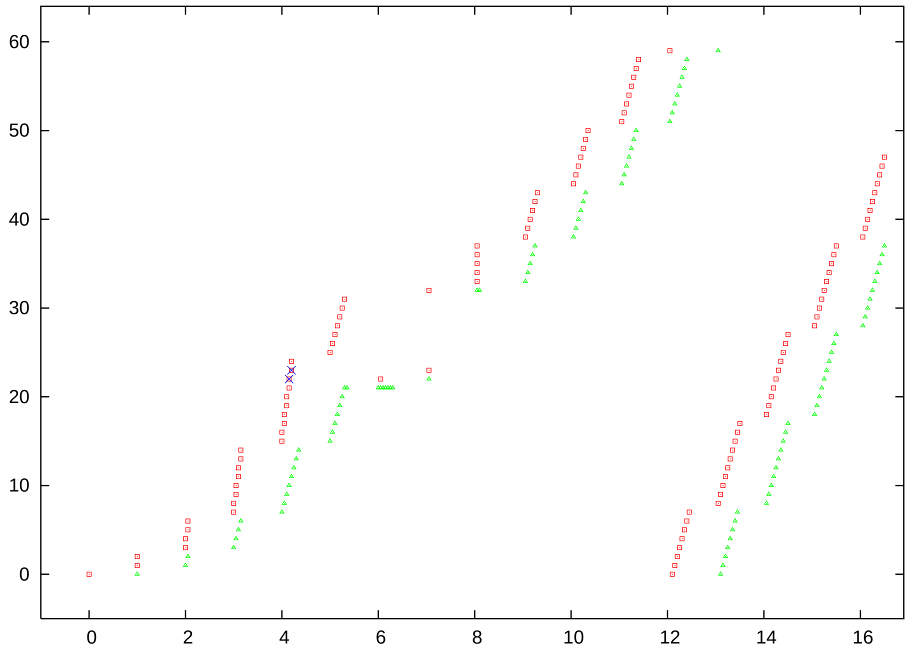


FIG. 5.1 – Algorithme *fast recovery* pour NewReno

NewReno ne retransmet qu'un paquet perdu par RTT, ce qui peut être très pénalisant lorsque l'on en perd beaucoup (par exemple lors du *slow start*). Néanmoins il est une bonne alternative à Reno car il est plus efficace tout en ne demandant pas énormément de changement.

Chapitre 6

TCP SACK

Ceci ne constitue pas exactement une nouvelle implémentation du protocole. C'est plutôt une amélioration qu'il est possible d'ajouter à diverses autres versions du protocole. Elle est décrite dans [2] et une extension est détaillée dans [5].

6.1 Selective acknowledgement

Généralement, les paquets sont envoyés par l'émetteur sous forme de rafales. Et il n'est pas inconcevable que plusieurs paquets soient perdus dans une même rafale. Comme TCP ne confirme que la réception des paquets contigus, la version Tahoe risque alors de renvoyer des paquets effectivement reçus, la version Reno va déclencher des time-out, et NewReno risque d'avoir besoin de plus de temps pour se remettre des pertes.

Dans tous les cas, parce que le schéma de confirmation n'est pas assez performant, il en résulte une dégradation des performances. La solution à ce problème est relativement simple : il suffit que le récepteur puisse dire à l'émetteur quels paquets ont effectivement été reçus. Cela n'est évidemment utile que lors de l'envoi de ACK dupliqués.

Ces informations supplémentaires sont passées grâce au champ option offert par le protocole TCP. À cause de la taille limitée de ce champ, le récepteur ne pourra renseigner l'émetteur que sur quatre blocs. Si jamais d'autres options du protocole sont utilisées, le nombre de blocs renseignés diminuera.

6.2 Duplicate SACK

C'est une amélioration de SACK proposée dans [5]. Cette fois, le récepteur a la possibilité de mentionner des paquets reçus plusieurs fois.

Supposons par exemple que l'émetteur n'a pas reçu de confirmation pour un paquet dans le délai imparti. Il va le supposer perdu et le réémettre après avoir modifié plusieurs de ses variables d'état pour refléter cette perte. Mais l'attente peut avoir été trop courte et le récepteur va alors recevoir deux paquets.

Quand l'émetteur recevra d'abord un ACK normal puis un ACK avec D-SACK, il saura que les paquets n'ont pas été perdu et qu'il a seulement besoin d'augmenter la temporisation. Il peut alors réinitialiser toutes ses variables à leurs valeurs d'avant l'incident et reprendre son rythme précédent.

6.3 Performances

Les figures 6.1 et 6.2 sont un exemple de ce qui se passe lors de la perte de trois paquets. NewReno réémet un des paquets perdus par RTT, sa phase de *fast recovery* va donc durer le temps de réémettre les trois paquets perdus. L'ajout de l'option SACK à NewReno lui permet de savoir quels paquets ont été effectivement perdus et de les réémettre dès que possible. Dans cet exemple, cela lui permet de sortir de sa phase de *fast recovery* un RTT plus tôt et au temps 17, 10 paquets supplémentaires ont pu être envoyés.

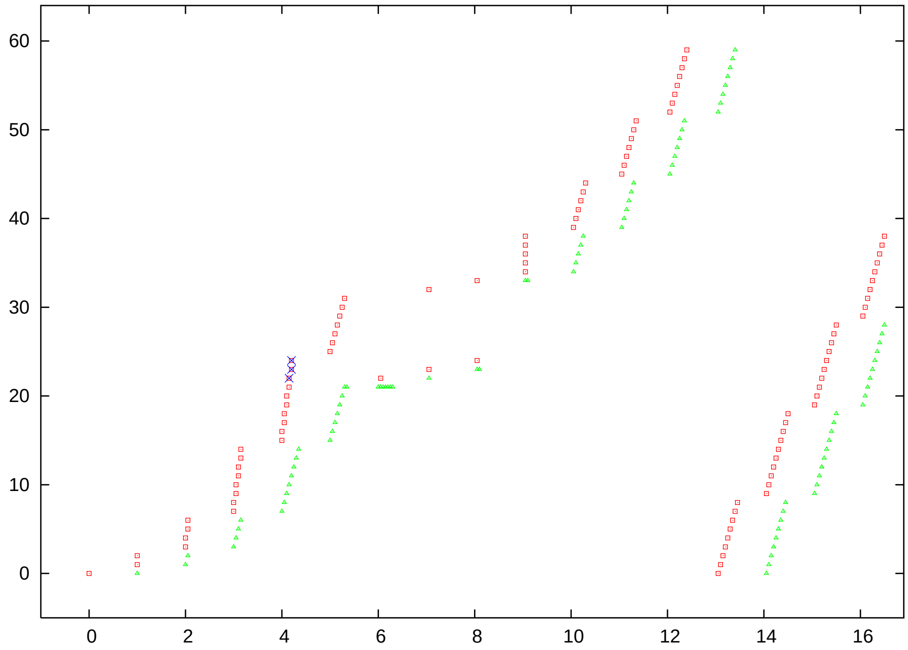


FIG. 6.1 – Perte de trois paquets sous NewReno

D'une manière plus générale, si les conditions ne sont pas trop mauvaises, NewReno réémettra un paquet perdu par RTT, alors que NewReno avec SACK sera capable de tous les réémettre d'un coup.

Cependant, SACK souffre de quelques faiblesses. D'une part, à cause de la taille limitée du champ option du protocole, le nombre de blocs SACK qu'il est possible de stocker est faible. Par conséquent, l'intérêt de SACK va diminuer si les paquets perdus ne sont pas contigus (par exemple si un routeur se débarrasse d'un paquet sur deux). D'autre part, le récepteur n'est pas obligé de conserver les paquets qu'il a confirmés par SACK (et non par ACK) indéfiniment. L'émetteur est donc obligé de réémettre ces paquets s'il y a expiration d'un temporisateur, même si le récepteur lui avait confirmé les avoir reçus la première fois.

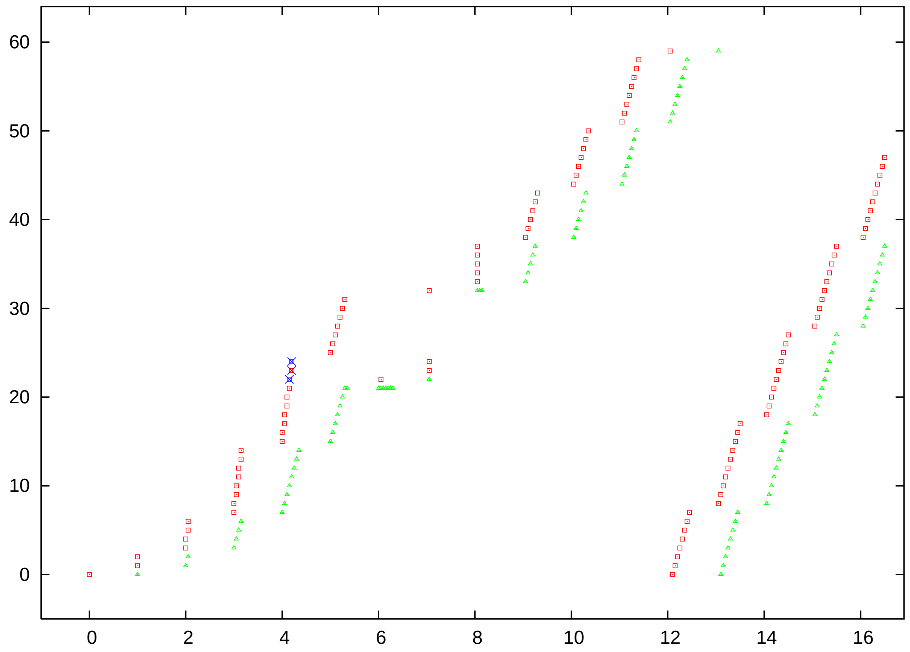


FIG. 6.2 – Perte de trois paquets sous SACK

Chapitre 7

TCP NetReno

Beaucoup de travail a été effectué dans les différentes implémentations de TCP afin de permettre à l'émetteur de se remettre d'une perte de paquet. Cependant, des mécanismes comme *fast retransmit* ou *fast recovery* ne sont pas toujours adaptés.

7.1 Small window

En effet, ces deux mécanismes ne se déclenchent qu'à la réception de trois ACK dupliqués. Malheureusement, pour que ces trois duplicata arrivent, il faut qu'un certain nombre de paquets soient envoyés (au moins cinq si l'on compte le paquet perdu). Par conséquent, ces mécanismes ne sont pas utilisés autant qu'on pourrait le souhaiter, si l'émetteur n'a rien à envoyer par exemple.

L'algorithme *small window* se met en marche dès la réception d'un ACK dupliqué. L'objectif n'est, cette fois, pas de retransmettre un paquet supposé perdu, mais de continuer la transmission des données à chaque réception de ACK. L'émetteur est autorisé à envoyer au plus deux paquets supplémentaires par rapport à la taille de la fenêtre de congestion.

7.2 Performances

Ce mécanisme est intéressant quand l'émetteur n'a que peu de données à envoyer ou quand sa fenêtre de congestion est petite, ce qui est le cas si le débit de son lien est faible. Comme le montre [8], la majorité des temps morts sont causés par un non-déclenchement des différents algorithmes de contrôles de flux. Et ces non-déclenchements sont d'autant plus présents que la fenêtre de congestion est petite.

Les figures 7.1 et 7.2 comparent le comportement de NetReno et des protocoles antérieurs quand la fenêtre de congestion est petite (dans cet exemple, 3 paquets) et qu'un paquet est perdu.

Parce que sa fenêtre de congestion est petite, l'émetteur ne recevra jamais assez de ACK pour passer en mode *fast recovery*. Par conséquent, pour les autres implémentations que NetReno, il faudra attendre l'expiration d'un délai pour que le transfert reprenne.

NetReno ne souffre pas de ce problème puisqu'il va continuer à envoyer des paquets à la réception des deux ACK dupliqués. Il obtiendra alors suffisamment de ACK pour passer en *fast recovery*.

Cela ne s'applique que s'il y a suffisamment de données à émettre. Dans le cas contraire, NetReno propose l'envoi de paquets vides afin de maintenir un flux régulier de ACK.

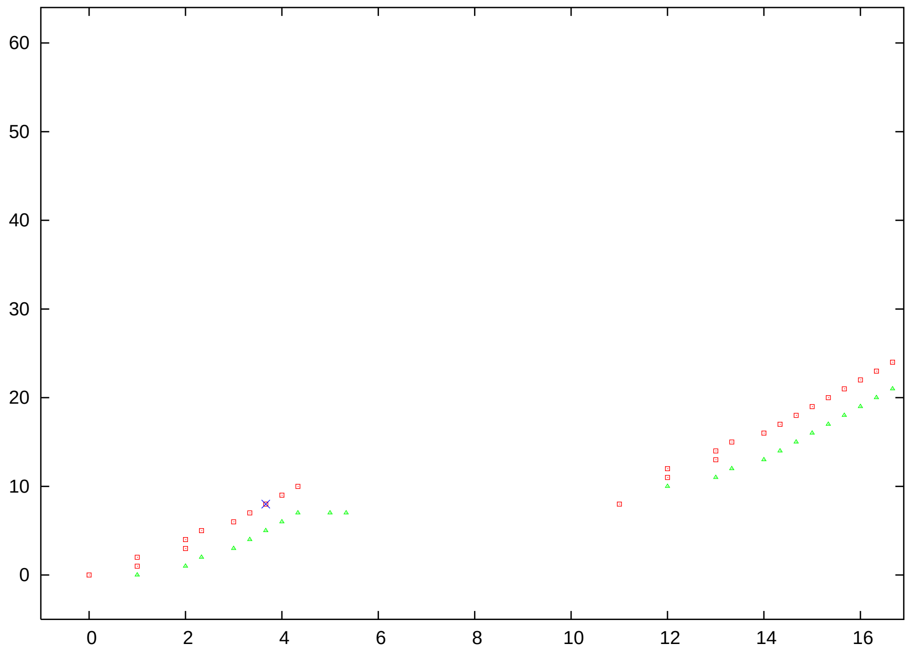


Fig. 7.1 – Perte d'un paquet et petite fenêtre

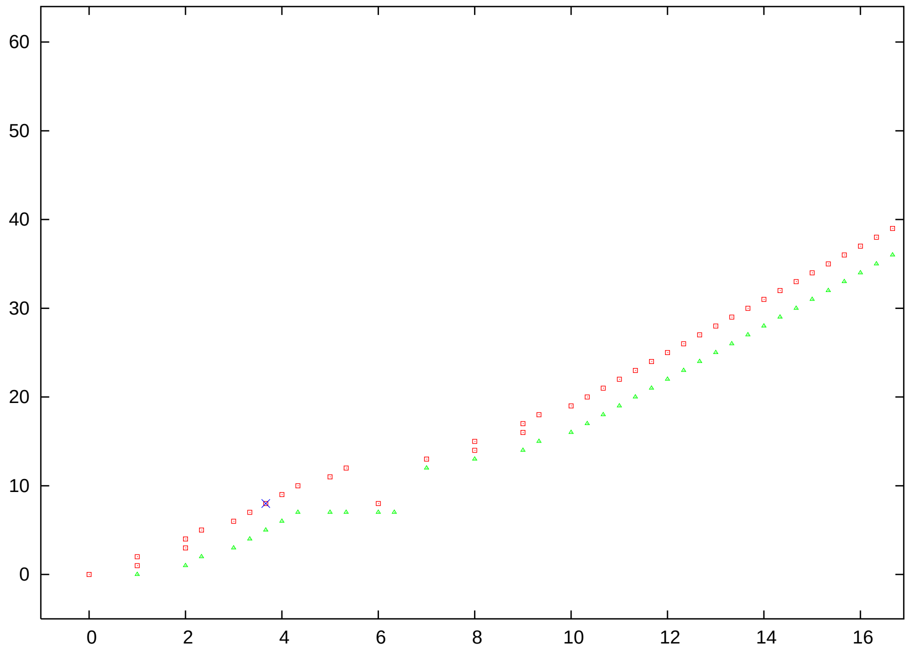


Fig. 7.2 – Perte d'un paquet et petite fenêtre sous NetReno

Chapitre 8

TCP Niagara

Niagara est une version très récente et très prometteuse de TCP. Elle a été développée par une équipe de l'Université de Toronto dirigée par K.L Eddy Law. Il y a encore assez peu d'informations disponibles sur le sujet, les documents utilisés ([10] et [11]) n'y font référence qu'indirectement.

Niagara est un dérivé de Reno pour le *slow start*, et de Vegas pour le contrôle de congestion, qui essaie d'en corriger les principaux défauts.

Niagara s'attaque au problème de la perte de paquets multiple lors du *slow start*. Ce qui pose problème dans Reno, c'est la valeur initiale de la variable *sstresh*. Si elle est trop petite, on mettra longtemps à prendre toute la bande passante disponible. Si elle est trop grande, on risque de perdre un grand nombre de paquets, et par conséquent (comme Reno est très mauvais pour rattraper les pertes de paquets multiples), on va perdre un temps énorme.

Niagara propose une méthode pour améliorer le *slow start*, qui demande plus de puissance de la part de la machine, mais elle est, d'après l'auteur, tout à fait exécutable sur les machines actuelles.

8.1 Window Bound et taux de changement de RTT

Couramment, on prend 64KB comme valeur de *sstresh*, mais des études montrent que sur les réseaux actuels cette valeur n'est pas toujours bien adaptée. Ici, on modifie ce *sstresh* – et d'ailleurs on lui donne un autre nom : *WIB* (*Window Bound*), plus adapté – pour lui donner une valeur plus significative. On se base sur l'observation que, en général, la fenêtre de congestion est plus petite que la fenêtre du récepteur. Des simulations semblent montrer que prendre pour *WIB* la moitié de la taille de la plus grande fenêtre de réception "publiée" par le destinataire est un bon compromis. On introduit le concept de RTT moyen pondéré : $saRTT_n$: la moyenne pondérée du RTT d'une connexion TCP pour le n-ème ACK reçu, avec le poids a. On note c le taux de changement de RTT par rapport à sa dernière valeur moyenne :

$$c = \frac{RTT - saRTT_{n-1}}{saRTT_{n-1}}$$

Une des nouveautés de cet algorithme est que l'on va permettre la transition de *slow start* à *congestion avoidance*, mais aussi le contraire : le passage de *congestion avoidance* à *slow start*. La valeur de c comparée à un seuil K va indiquer quel mécanisme de contrôle de flux utiliser, *slow start* ou *congestion avoidance* (il est recommandé de prendre $K = 0.5$).

- En *slow start* :
  - Si $c > K$ et $cwnd < WIB$, on reste en *slow start*.
  - Sinon, on passe en *congestion avoidance*.

- En *congestion avoidance* :
  - Si $c < -K$ et $cwnd < WIB$, on passe en *slow start*.
  - Sinon, on reste en *congestion avoidance*.

8.2 Congestion Avoidance

Niagara reprend le mécanisme de détection de congestion de Vegas, en y corrigeant le principal défaut. Dans Vegas, on prenait comme valeur pour $baseRTT$, le maximum des RTT depuis le début de la connexion, ce qui est absurde : si jamais une nouvelle connexion Vegas se crée à côté de la première, elle obtiendra fatalement une plus grande bande passante. Pour rétablir un équilibre entre les connexions (et en particulier pour résister à des algorithme de contrôle de flux “agressifs” comme ceux de Tahoe, Reno, NewReno...), on prend dans Niagara comme $baseRTT$ le RTT pondéré moyen ($saRTT_n$) vu précédemment, ce qui corrige effectivement le problème.

8.3 Comparaison avec Reno et Vegas

Au niveau du contrôle de flux, une connexion Vegas seule était plus efficace qu'une connexion Reno, puisqu'elle arrivait à prendre effectivement toute la bande passante en permanence, donc *a fortiori* Niagara fait aussi bien que cela. Niagara est un protocole très récent, et il n'y a encore malheureusement qu'assez peu de documentation disponible, et encore moins de comparatifs (il n'existe même pas encore d'implémentation). Nous avons tout de même trouvé ce comparatif (réalisé par les auteurs de Niagara) : le comportement de deux connexions Reno (figure 8.1) comparé au comportement de quatre connexions Niagara (figure 8.2).

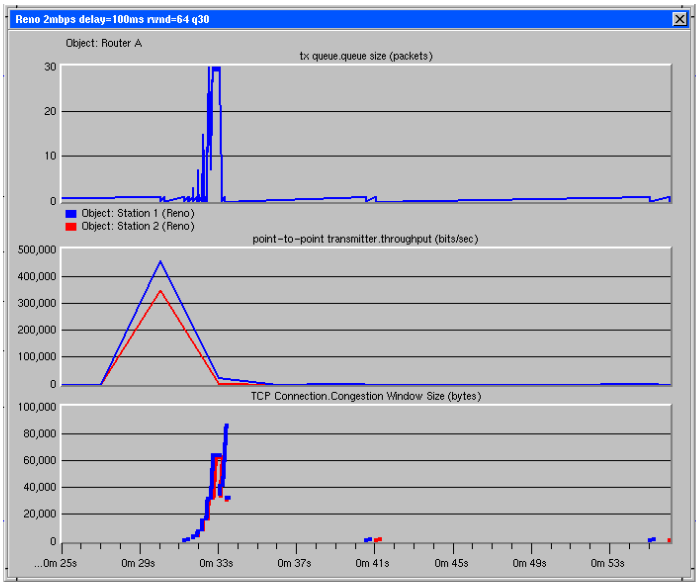


FIG. 8.1 – Comportement des deux connexions Reno

On constate qu'il n'y a plus de problème d'équité lorsqu'il y a plusieurs connexions, Niagara fait donc mieux que Vegas. On voit aussi que les connexions Reno ont énormément de problème à cause d'un *slow start* où un très grand nombre de paquets ont été perdus.

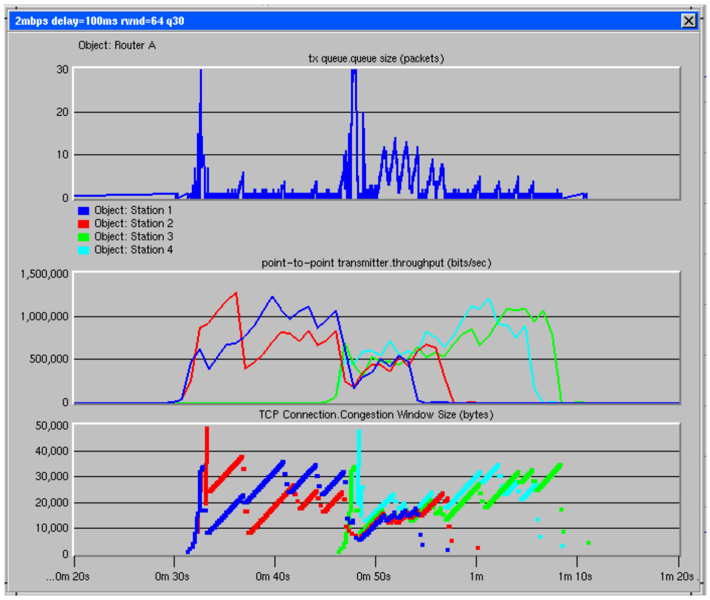


FIG. 8.2 – Comportement des quatre connexions Niagara

Conclusion

Les versions présentées ici ne forment qu'une partie de toutes les recherches effectuées. Par exemple, les ECN (*Explicit Congestion Notification*) peuvent être ajoutées au protocole IP pour améliorer l'anticipation des congestions. Une autre amélioration de TCP peut être réalisée par l'intermédiaire de l'option Timestamp.

La version actuellement la plus utilisée est NewReno. C'est une variante de Reno capable de mieux gérer la perte de plusieurs paquets. Les noyaux Linux récents utilisent par exemple une version de NewReno capable de gérer l'extension SACK (et D-SACK) ainsi que les notifications ECN.

Il est cependant fort probable que les prochaines versions mettront en application les avancées faites avec des implémentations comme NetReno ou Niagara, ou d'autres...

Bibliographie

[1] *RFC 793, Transmission Control Protocol*

[2] *RFC 2018, TCP Selective Acknowledgment Options*

[3] *RFC 2581, TCP Congestion Control*

[4] *RFC 2582, The NewReno Modification to TCP's Fast Recovery Algorithm*

[5] *RFC 2883, An Extension to the SACK Option for TCP*

[6] L.S. Brakmo, S.W. O'Malley, L.L. Perterson, *TCP Vegas : New Techniques for Congestion Detection and Avoidance*, 1995

[7] *SSF TCP Regression Tests*, http ://www.ssfnet.org/Exchange/tcp/tcpTestPage.html

[8] Dong Lin, H.T. Kung, *TCP Fast Recovery Strategy : Analysis and Improvements*

[9] Kevin Fall, Sally Floyd, *Simulation-based Comparisons of Tahoe, Reno and SACK TCP*

[10] K.L. Eddie Law, Wing-Chung Hung, *Problems Investigation in TCP Flow Control Mechanisms*, Biennal Symposium 2002 (?)

[11] K.L. Eddie Law, Wing-Chung Hung, *Problems and Solutions for the TCP Slow-Start Process*