MASTER EN SCIENCES INFORMATIQUES

TCP Xeno
interaction de GTCP et de TCP Veno

Styliaras, Christos

Award date:
2007

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique

Année académique 2006-2007

TCP Xeno

Interaction de GTCP et de TCP Veno

Christos Styliaras

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique

Septembre 2007

Résumé

Ces dernières années on assiste à une croissance quasi exponentielle du réseau Internet. Les réseaux sans fil, les réseaux cellulaires, les applications vidéo et voix sur IP ou les contrôles industriels, exigent du service best-effort offert par le réseau un débit de plus en plus élevé et constant et des garanties de largeur de bande. TCP a subi beaucoup d'améliorations depuis sa conception en vue d'optimiser la qualité du trafic de données. Parmi ces optimisations, Veno a été conçu pour le service best-effort. De son côté, GTCP est un des rares protocoles TCP qui assure un service de débit garanti. Ce mémoire est une étude de la pertinence de la combinaison de ces deux mécanismes. Cette étude a comme résultat un nouvel algorithme, TCP Xeno. Il a été développé avec les moyens qu'un système ouvert comme Linux peut offrir. Des tests de validation de comportement et de performance avec les algorithmes originaux ont été mis au point et effectués à l'aide du simulateur de réseau netem et de l'analyseur de trafic réseau Ethereal. Les résultats de ces tests ont montré que TCP Xeno offre de meilleures performances que TCP Reno et TCP Veno.

Abstract

In recent years we have observed a quasi-exponential growth of the Internet network. The wireless, as well as the cellular (mobile communication) networks, the video and voice applications on IP or industrial controls, require the "best-effort" service provided by the network, in addition to an increasingly high and constant flow, as well as bandwidth guarantees. TCP underwent many improvements since its design, in order to optimise the quality of data traffic. Among these optimisations, Veno was conceived for the "best-effort" service. GTCP is one of the rare TCP protocols, which ensure a service of guaranteed flow. This final year project is a study of the relevance of the combination of these two mechanisms. As a result of this study, a new algorithm, TCP Xeno was created. It was developed with the means that an open system like Linux can offer. Validation behavioural tests, plus performance tests with the original algorithms were developed. These tests were carried out using the "netem" network simulator, as well as the "Ethereal" network traffic analyser. The results of these tests showed that TCP Xeno behaves as predicted and that in the event of congestion, it offers better transfer times than TCP Reno and TCP Veno.

Avant-propos

Au terme d'un travail de longue haleine, il est de tradition de remercier ceux qui directement ou indirectement y ont participé.

Ce mémoire est l'aboutissement d'une aventure qui a commencé, un peu par hasard lorsque poussé par la volonté d'acquérir des compétences nouvelles, j'ai franchi la porte de la LIHD.

Je remercie le Professeur Laurent Schumacher d'avoir accepté de me superviser et me soutenir pendant la longue période de préparation de ce travail.

Je remercie M. Cédric Boudin pour la patience qu'il a montrée en essayant de m'initier aux subtilités de Linux. Un grand merci aussi à MM. André Lemal, Michel Kempeneers et Fabrizio Marinaro pour leur aide pendant l'élaboration de ce document et à M. Georges C. pour le prêt du matériel nécessaire à la mise au point de l'algorithme.

Enfin, je tiens à remercier tous ceux qui, bien que n'étant pas directement impliqués dans ce travail, m'ont soutenu d'une manière ou d'une autre au cours de cette période.

Christos

Glossaire

ACK (*acknowledgement*) : acquittement, accusé de réception cumulatif.

ACK partiel (*acquittement cumulatif partiel*) : accusé de réception qui n'acquitte qu'une partie de tous les segments déjà envoyés.

AIMD (*Additive Increase Multiplicative Decrease*) : algorithmes utilisés par le mécanisme de congestion avoidance.

API (*Application Programming Interface*): interface de programmation qui permet de définir la manière dont un processus informatique peut communiquer avec un autre.

best-effort : service fourni par TCP dont le rôle est de faire de son mieux (sans réservation des ressources) pour amener les paquets à destination.

cwnd (*congestion window*) : fenêtre de congestion, variable TCP du contrôle de congestion qui limite le nombre de segments qui peuvent être envoyés en fonction de l'état du réseau.

dupACK : accusé de réception dupliqué.

ECN (*Explicit Congestion Notification*): mécanisme permettant aux routeurs compatibles de signaler une congestion avant la perte de paquets [4].

End-to-end (*E2E, bout-à-bout*): principe selon lequel chaque extrémité d'une connexion est responsable du contrôle du débit des données qu'elle envoie.

fullACK (*acquittement cumulatif global*) : accusé de réception dont le numéro de séquence permet d'acquitter tous les segments déjà envoyés.

IETF (*Internet Engineering Task Force*): groupe informel, international, ouvert à tout individu, qui participe à l'élaboration de standards pour Internet. L'IETF produit la plupart des nouveaux standards d'Internet.

IP (*Internet Protocol*) : protocole de gestion de la couche réseau.

IPv4 (*Internet Protocol version 4*) : première version d'IP à avoir été largement déployée, et qui constitue encore la base de l'Internet. Elle est décrite dans la RFC numéro 791.

IPv6 (*Internet Protocol version 6*) : successeur du protocole IPv4. Une adresse IPv6 est longue de 16 octets, soit 128 bits, contre 4 octets (32 bits) pour IPv4.

LIFO (*Last In, First Out*) : principe de la pile, dernier arrivé, premier sorti.

MSS (*maximum segment size*) : taille maximum d'un segment, valeur négociée au moment de la connexion, entre l'émetteur et le récepteur. Sa valeur maximale est 1456 bytes.

maximum sequence number : le numéro de séquence le plus élevé de segments qui ont été émis sur le réseau.

OSI (*Open Systems Interconnection*) : l'OSI est une initiative de normalisation de la gestion des réseaux qui a commencé en 1982 par l'International Standard Organization (ISO).

QoS (*Quality of Service*) : la Qualité de Service est la capacité à véhiculer dans de bonnes conditions un type de trafic donné, en termes de disponibilité, débit, délais de transit, taux de perte de paquets. Son but est ainsi d'optimiser les ressources du réseau et de garantir de bonnes performances aux applications critiques.

RFC (*Request For Comments*) : demande de commentaires. Série de documents et normes concernant l'Internet, commencée en 1969. Peu de RFC sont des standards, mais tous les standards de l'Internet sont enregistrés en tant que RFC.

RTO (*Recovery Timeout*) : la durée maximale d'interruption admissible constitue le temps maximal acceptable durant lequel une ressource informatique peut ne pas être fonctionnelle après une interruption majeure de service.

RTT (*Round-Trip-Time*) : temps écoulé entre l'envoi d'un paquet et son acquittement.

SMSS (*Sender Maximum Segment Size*) : nombre maximal d'octets qui peuvent être envoyés dans un simple segment TCP sans l'en-tête (header).

ssthresh (*slow-start threshold*) : variable TCP du contrôle de congestion qui représente le seuil de la taille de la fenêtre de congestion cwnd pour passer de l'état de contrôle de congestion slow-start à l'état congestion avoidance.

Timer (*temporisateur*) : mécanisme de mesure du temps écoulé depuis l'émission d'un paquet. A l'expiration de ce temps (voir RTO), le mécanisme déclenche un timeout.

Timeout : expiration du temporisateur déclenché par l'envoi d'un segment à l'autre extrémité de la connexion.

Table des matières

Chapitre 1

Introduction

En quelques années, Internet est devenu un composant essentiel des systèmes de communication : un nombre croissant d'applications reposent sur l'accessibilité et la rapidité des réseaux et la fiabilité de leur infrastructure. Dans ce contexte, il est important de comprendre le fonctionnement des protocoles et acteurs principaux qui assurent le maintien de la fiabilité du système.

Bien que mis en oeuvre depuis plus de trente ans, le protocole de transport TCP fait toujours l'objet d'optimisations. Le présent mémoire en aborde deux, GTCP [2] et TCP Veno [1].

GTCP est une modification de l'émetteur TCP destinée à accroître le taux de transfert global, en scindant la fenêtre de congestion en deux sous-fenêtres dont les tailles évoluent de manière différente. De son côté, TCP Veno est une variante de TCP Reno qui tente d'éviter la réduction de la fenêtre de congestion quand les pertes de paquet ne sont pas dues à la congestion. Ses promoteurs affirment qu'elle peut atteindre un taux de transfert supérieur de 30% à celui de TCP Reno lorsque le lien physique est de piètre qualité, par exemple dans le cas des réseaux d'accès radio.

L'objet du mémoire est l'étude de la pertinence de la combinaison de ces deux mécanismes, dans l'optique de connexions TCP à débit minimal garanti dans les réseaux d'accès radio, pour les applications de streaming par exemple.

Les trois premiers chapitres du document comprennent une présentation du TCP, une description du problème de la congestion et des différents mécanismes de contrôle de la congestion développés pour ce protocole, ainsi que la présentation de Veno et de GTCP.

La suite du document est organisée comme suit : le chapitre 4 étudie les avantages de deux protocoles, TCP Veno et GTCP, et motive la conception de TCP Xeno. Le chapitre 5 explique le mécanisme de gestion de la congestion dans Linux. Il commence par une présentation des structures de données et des variables utilisées par ce mécanisme. Ensuite, il continue avec une présentation de l'interface qui permet d'implémenter des modules de contrôle de congestion dans le kernel de Linux et termine par les conclusions concernant l'implémentation du module.

Le chapitre 6, est une présentation des dispositifs choisis pour évaluer le module Xeno et une discussion sur les tests de validation et de performance et les résultats obtenus. Le document se termine par la conclusion de cette étude.

Chapitre 2

Le protocole TCP

2.1 Introduction

Ainsi que l'exprime Constantinos Makassikis dans son mémoire [29] :

« TCP est un protocole de communication réseau qui se situe au niveau de la couche transport (couche 4 du modèle OSI) et dont l'objectif principal est de fournir à la couche application (couche 7 du modèle OSI) un moyen fiable de transmission des données. En effet au niveau le plus bas, les données ne sont pas à l'abri des erreurs et de corruption liées aux supports de transmission, aux équipements réseau ou encore à la charge du réseau. De plus, comme le choix des routes (routage) s'effectue dynamiquement, des données issues d'un même émetteur et destinées à un hôte commun n'empruntent pas toujours le même chemin pour l'atteindre. Il s'ensuit que des délais d'acheminement peuvent varier de manière importante et donner lieu à des arrivées des données en désordre, ou à des duplications. De plus, la transmission de données nécessite, suivant les équipements utilisés, de prendre en considération et donc de gérer un certain nombre de paramètres supplémentaires tels que la taille optimale des données à envoyer.

En réponse à ces difficultés, TCP offre une interface commune à la couche application permettant de faire abstraction de la plupart des considérations liées à la transmission fiable et optimale des données. Par ailleurs, en plus du contrôle d'erreur, TCP assure également un contrôle de flux et de congestion : selon la capacité du récepteur à recevoir et consommer les données qui lui sont envoyées ainsi que la charge du réseau, TCP adaptera la vitesse de transmission. »

Les spécifications de fonctionnement de TCP sont décrites dans la RFC 793 [16]. Les algorithmes de base pour effectuer le contrôle de la congestion ont été conçues par Van Jacobson [18]. Plus tard, ces algorithmes de contrôle ont été inclus dans la spécification de normes TCP par l'IETF [22] [5] [6] [7].

2.2 Transmission orientée connexion et fiable

En TCP, le fait d'être en mode connecté signifie que les deux extrémités qui échangent des données vérifient continuellement leur existence réciproque.

Le rôle de TCP est de fournir un service de flux d'octets orienté connexion et fiable [9]. Pour assurer une transmission fiable de données, TCP utilise une technique d'acquittements avec retransmission qui consiste à envoyer un paquet, puis à attendre de la part du récepteur un acquittement qui confirme la bonne réception avant d'émettre le paquet suivant.

La détection d'une perte (la corruption aussi) de paquet de données repose sur l'utilisation d'un temporisateur qui est déclenché après l'envoi d'un paquet. L'expiration du temporisateur avant la réception de l'acquittement est interprété comme la perte du paquet et entraîne sa retransmission.

Le mécanisme de TCP qui assure la bonne réception des paquets dans leur intégrité et dans le bon ordre peut être détaillé comme suit [9]:

- Les données sont fractionnées en **segments** dont la taille est jugée la meilleure par TCP pour l'émission.

- Lorsque TCP émet un segment, il déclenche un **temporisateur** (**timer**) attendant de l'autre extrémité une information sur la bonne réception de ce segment.

- Lorsque TCP reçoit des données de l'autre extrémité de la connexion, il émet un **acquittement** ou **accusé de réception** (**ACK**).

- TCP maintient une somme de contrôle (**checksum**) dans son en-tête de données, d'extrémité à extrémité (E2E), qui lui sert à détecter toute modification des données en transit. Si un segment parvient avec un checksum invalide, le paquet est rejeté et il n'est pas acquitté par le récepteur.

- Puisque les segments peuvent arriver de la couche IP en désordre, TCP réordonne les données si nécessaire, passant les données reçues dans l'ordre correct à l'application.

- TCP rejette les segments dupliqués à la réception.

- TCP fournit aussi un contrôle de flux. Pour cela il utilise à chaque extrémité un buffer de taille finie (protocole des fenêtres glissantes). Une connexion TCP permet aussi à l'autre extrémité d'émettre autant de données que peuvent en recevoir les buffers de son correspondant.

2.2.1 Le contrôle de flux

Le protocole des fenêtres glissantes

Le but de ce protocole, appelé aussi **fenêtre offerte** [8] est d'optimiser la transmission des données entre les deux extrémités. En regardant la figure 1 on constate qu'après l'envoi d'un nouveau segment, l'attente d'un acquittement avant de pouvoir envoyer un autre segment, limite l'utilisation de la totalité de la capacité du réseau. L'amélioration que le protocole des fenêtres glissantes apporte est de pouvoir envoyer plusieurs paquets sans se préoccuper de l'acquittement des paquets déjà envoyés. De cette façon, la bande passante disponible est pleinement exploitée.

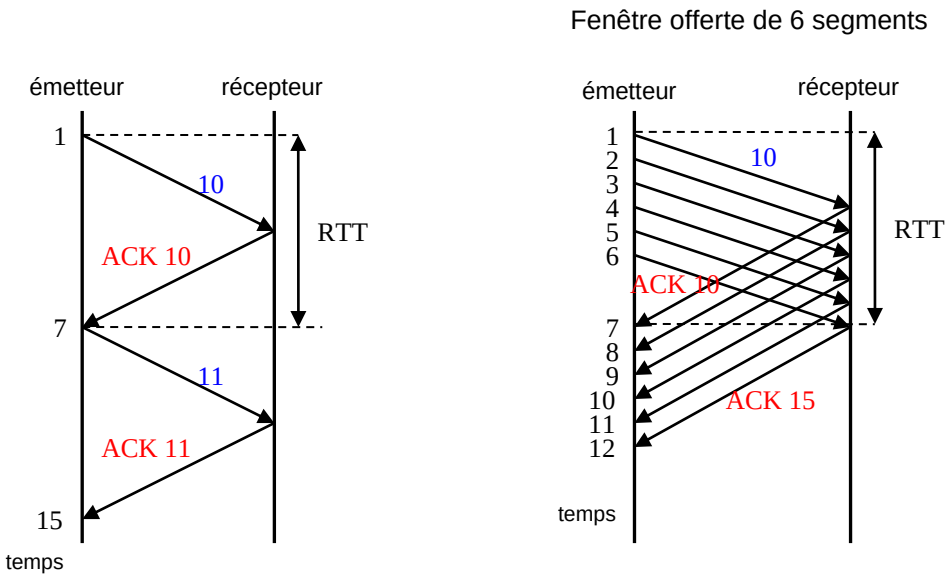


Figure 1 : La fenêtre glissante optimise le flux de données

La fenêtre offerte permet de délimiter le tampon d'émission en trois zones [29] (figure 2) :

- En amont du bord gauche de la fenêtre se trouvent les données qui ont été acquittées par le récepteur.
- A l'intérieur de la fenêtre se trouvent les données déjà émises mais non acquittés.
- En aval du bord droit de la fenêtre se trouvent les données prêtes à être envoyées.

A chaque réception d'un acquittement, le bord gauche de la fenêtre se déplace vers la droite.
A chaque envoi d'un nouveau segment, le bord droit de la fenêtre se déplace à droite.

Par définition un acquittement est le numéro du premier segment attendu par le récepteur. On peut déduire que tous les numéros de paquets précédant ce numéro sont déjà reçus par le récepteur. On peut alors parler d'un **acquittement cumulatif**.

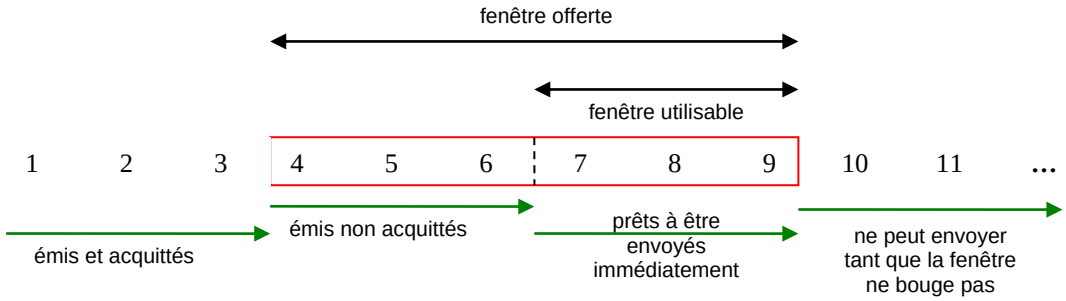


Figure 2 : Visualisation de la fenêtre glissante de TCP [8]

Le mécanisme de contrôle de flux avec le protocole de fenêtres glissantes permet d'informer l'émetteur de la quantité de place disponible sur le tampon du récepteur afin qu'il puisse faire varier en conséquence la taille de sa fenêtre.

2.2.2 Le contrôle de la congestion

Le principal défaut de TCP est sa réaction a posteriori à des symptômes résultants d'une congestion. Dans des liaison simples de réseau local sans nœuds, l'émetteur envoie sans problème de multiples segments jusqu`à ce que la taille de la fenêtre indiquée par le destinataire soit atteinte. En revanche, lorsqu'il y a des nœuds (p.ex. des routeurs) et des liens de plus faible débit entre les deux machines, certains paquets peuvent être mis en file d'attente par les routeurs ce qui entraîne des pertes. Le débit peut alors être réduit de manière significative.

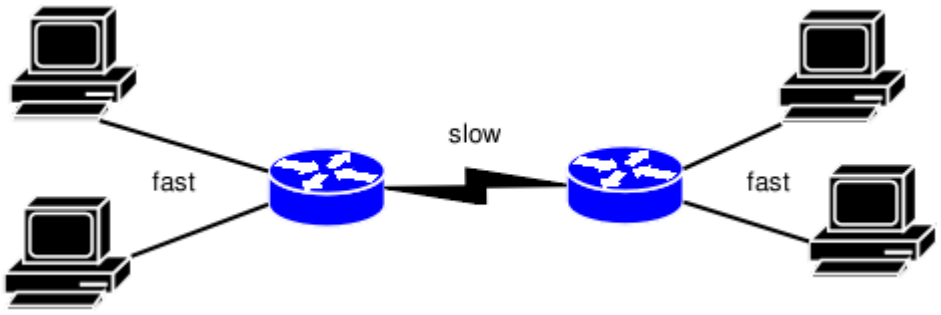


Figure 3: Testing congestion situations, Linux gazette [12]

D'une manière générale, quand la charge est plus importante que ce qu'un nœud est en mesure de traiter, il se produit une congestion. Pour éviter ce phénomène il faut alors adapter à chaque instant la bande passante disponible.

Le premier stade de la gestion d'une congestion est sa détection. Ainsi, lorsqu'un temporisateur de retransmission expire on peut en déduire qu'il y a congestion. La perte d'un paquet est la seule information sur l'état du réseau [10].

TCP utilise un certain nombre de mécanismes afin d'obtenir une bonne robustesse et des performances élevées. L'idée de ces mécanismes est d'arriver à une situation dans laquelle l'émetteur injecte un nouveau paquet dans le réseau simultanément à la réception d'un paquet par le destinataire. La congestion est gérée uniquement par l'émetteur, le récepteur ne fait qu'envoyer des accusés de réception.

Les mécanismes de contrôle de la congestion

Les travaux de Van Jacobson en 1988 [18] ont abouti à la proposition de deux algorithmes. Ces derniers permettent à TCP de réguler le débit des paquets en fonction de l'état du réseau. Il s'agit de **slow-start** et de **congestion avoidance**.

Dans ces deux algorithmes l'émetteur utilise une fenêtre appelée **fenêtre de congestion** ou **cwnd**. Alors que la fenêtre glissante est un contrôle de flux imposé par le récepteur, la fenêtre de congestion est le contrôle de flux imposé par l'émetteur. Sa taille varie en fonction de l'état de la connexion.

L'approche consiste à augmenter la taille de la fenêtre de congestion tant que le réseau le permet, puis à la réduire en cas de congestion. Il s'agit du mécanisme **AIMD** (**Additive Increase Multiplicative Decrease**). La phase Additive Increase contient deux niveaux, le slow-start (ou démarrage lent) et le congestion avoidance (évitement de la congestion).

Bien que les deux algorithmes, congestion avoidance et slow-start, soient indépendants, ils sont généralement implémentés ensemble. Ils nécessitent, outre la fenêtre de congestion un seuil associé au slow-start appelé **ssthresh**.

La phase de démarrage lent (slow start)

Le but de cet algorithme est de retrouver rapidement la bande passante disponible.

Pendant la phase du slow-start, l'augmentation de la fenêtre de congestion suit une croissance exponentielle. A la réception d'un acquittement de l'autre extrémité la taille de la fenêtre augmente de la manière suivante :

- L'émetteur transmet un segment.
- Quand l'accusé de réception est reçu, la fenêtre de congestion est incrémentée de 1 à 2, et 2 segments sont envoyés.
- Lorsque un ACK arrive pour chacun de ces segments, la fenêtre est incrémentée de 2 à 4 segments et ainsi de suite ... jusqu'à la fin du transfert ou jusqu'à ce que le buffer de l'émetteur soit plein ou encore jusqu'à ce que la congestion survienne.

Ainsi, la taille de la fenêtre double de taille après chaque RTT.

Le slow-start se poursuit jusqu'à ce que la fenêtre de congestion atteint le ssthresh. Une fois que la taille de la fenêtre dépasse le sshtresh l'algorithme passe en phase d'évitement de congestion.

La phase d'évitement de la congestion

L'idée de l'algorithme est qu'une indication de pertes de paquets signale une situation de congestion.

Pendant cette phase l'accroissement de la fenêtre de congestion est linéaire. Elle augmente à raison d'un segment par RTT. Pour être précis la congestion avoidance impose que cwnd soit incrémentée de $^{1}/_{cwnd}$, plus une petite fraction de la taille du segment (la taille du segment divisée par 8) à chaque fois qu'un ACK est reçu [9].

En cas d'indication de perte de paquets ou à l'expiration du temporisateur associé à chaque paquet, la fenêtre de congestion est réduite à un minimum prévu. TCP peut ainsi ralentir son débit d'émission de paquets. Ensuite l'algorithme passe en phase slow-start.

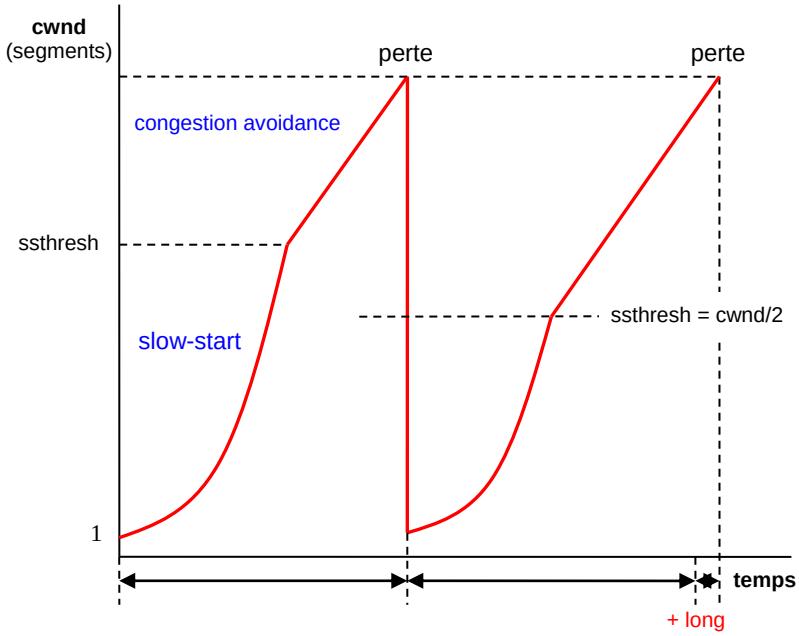


Figure 4 : Visualisation du slow-start et de congestion avoidance

Dans les années 1990, des améliorations de l'algorithme de congestion ont été proposées, dont Fast Recovery et Fast Retransmit [5][6][7] .

Fast Retransmit

TCP est conçu pour générer un **acquittement dupliqué** ou **dupACK** lorsqu'un segment est reçu en désordre [9]. Ainsi, l'autre extrémité peut savoir qu'un segment a été reçu en désordre et est informée du numéro de séquence manquant.

Quand TCP émet des acquittements, dans le cas de paquets désordonnés (pas dans la bonne séquence), ceux-ci contiennent toujours le numéro du plus ancien paquet non encore acquitté. Il devient alors possible pour TCP de recevoir plusieurs fois le même acquittement. Pour TCP, « cette réception est une indication ambiguë, étant donné qu'il peut s'agir d'un mauvais ordonnancement des paquets ou d'une perte. Pour éliminer ce problème de mauvaise interprétation **Fast Retransmit** a été implémenté » [29].

TCP est contraint d'attendre 3 acquittements dupliqués (**3dupACK**) successifs avant de considérer qu'un segment est perdu. Le segment considéré comme perdu est retransmis sans attendre l'expiration du temporisateur.

Voici l'algorithme, comme il est défini dans [5] :

1. A la réception d'un troisième dupACK,

```
sshtresh = max ( FlightSize / 2, 2*SMSS )
```

où FlightSize est le *nombre de paquets émis mais pas encore acquittés* [5].

2. Retransmettre le segment demandé avant que le temporisateur de la retransmission n'expire et faire :

cwnd = sshtresh + 3.

L'algorithme considère ainsi que le 3 dupACK successifs ont été déclenchés par la réception à l'autre extrémité de 3 nouveaux segments. Cette dernière opération est le « **gonflement** » de la fenêtre de congestion [5].

3. Passer en phase Fast Recovery

Fast Recovery

Après Fast Retransmit, TCP passe en **Fast Recovery**. Le but de ce mécanisme est de maintenir le flux de paquets sur le réseau [5].

- Pour chaque dupACK recu, l'émetteur envoie un nouveau segment, si cela est permis par la taille de la fenêtre offerte par le récepteur. La fenêtre de congestion est incrémentée de 1.

- A la réception d'un acquittement pour le segment perdu, la fenêtre est réduite à la valeur du ssthresh et passe en phase de congestion avoidance, sinon à l'expiration du temporisateur (timeout), TCP considère qu'il y a perte sévère, la fenêtre de congestion cwnd est ramenée à sa valeur initiale et TCP repart en mode slow-start.

En examinant la représentation graphique du Fast Recovery à la figure 5, on constate qu'en comparant avec le comportement slow-start - congestion avoidance - slow-start représenté par la figure 4, TCP atteint beaucoup plus rapidement son débit optimal.

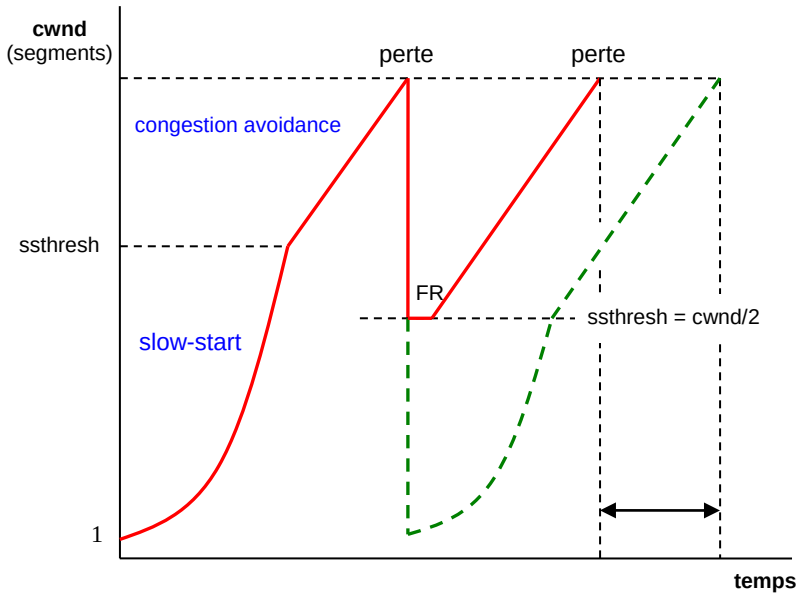


Figure 5 : Visualisation du Fast Recovery (FR)

Retransmission Timeout (RTO)

L'émetteur TCP détermine le **retransmission timeout** en fonction de la mesure des temps aller-retour (round-trip-time, RTT) des paquets. Quand un segment arrive à l'émetteur TCP, les spécifications RFC [16] lui demandent d'ajuster la valeur de RTO comme suit [4] :

$$SRTT = \frac{7}{8} SRTT + \frac{1}{8} RTT$$

$$RTTVAR = \frac{3}{4} RTTVAR + \frac{1}{4}(SRTT - RTT)$$

$$RTO = \max (SRTT + 4\ RTTVAR, 1) \text{ secondes}$$

où RTT est le round-trip-time mesuré, RTTVAR est la variation des RTT et SRTT (smooth RTT) est une moyenne des RTT ou RTT « corrigé ».

Le retransmission timeout est considéré comme une indication de perte et l'émetteur déclenche la retransmission des segments non acquittés. En outre, quand un RTO se produit, l'émetteur initialise la fenêtre de congestion à un segment, puisque le RTO indique que la charge de réseau doit diminuer nettement. Ensuite il entre en phase slow-start, de manière à retrouver rapidement la bande passante disponible.

Chapitre 3

L'évolution des algorithmes de contrôle de congestion

Le protocole de transport de TCP a progressé lentement depuis sa conception en 1988. Cette page présente les réalisations principales, décrivant les algorithmes et les différences entre chaque version.

3.1 TCP Tahoe

Première version de TCP proposée par Van Jacobson [18]. Ses phases successives sont :

- slow-start
- Congestion Avoidance
- Fast Retransmit

Fast Restransmit est déclenché à la réception de 3 dupACK consécutifs. Son grand défaut est que même si le réseau n'est pas chargé, il effectue un slow-start, réduisant brutalement le débit [13]. La figure 6 présente les états possibles de l'algorithme TCP Tahoe :

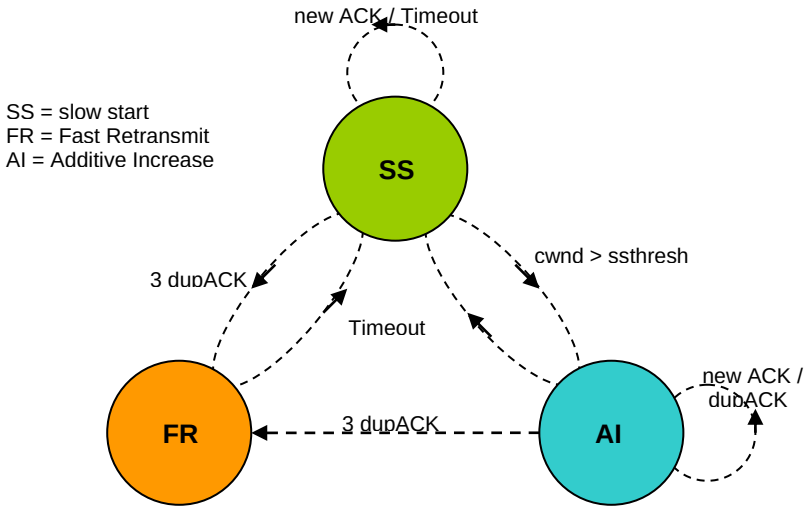


Figure 6 : états de TCP Tahoe

3.2 TCP Reno

Version améliorée de TCP Tahoe avec l'introduction de l'algorithme Fast Recovery après un Fast Retransmit. Cela permet de résoudre le défaut principal de Tahoe. Ses phases successives sont donc :

- slow-start
- Congestion Avoidance
- Fast Retransmit
- Fast Recovery

Fast Retransmit : dans Reno lors de la détection d'une perte par 3dupACK, seul le paquet supposé perdu est retransmis et pas les suivants.

Fast Recovery : on ne repart pas en slow-start après le Fast Retransmit [5].

La figure 7 présente les états possibles de l'algorithme TCP Reno :

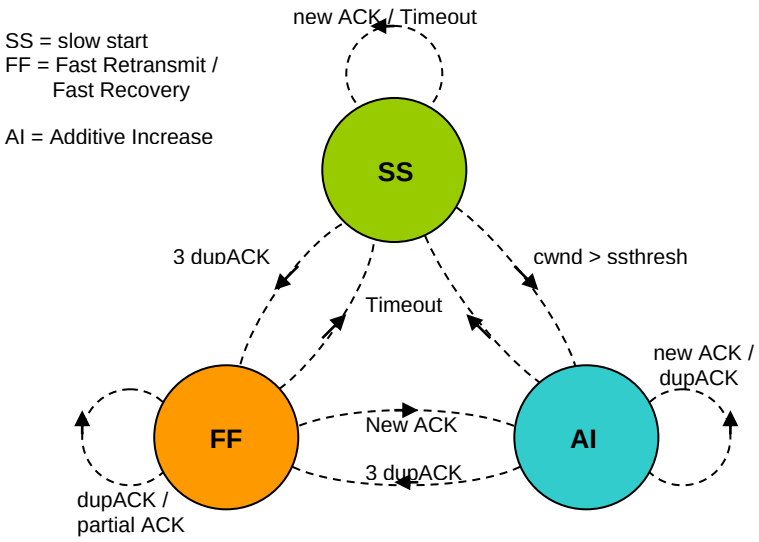


Figure 7 : états de TCP Reno

Problème de **"successive fast retransmits"** [13][17] :

Les retransmissions successives au sein d'une même fenêtre de données peuvent mener à un blocage.

En effet, Reno considère les acquittements partiels comme des dupACK. Après un deuxième Retransmit dans la même fenêtre, il y a des fortes chances que Reno reçoive un ACK partiel. Dans ce cas, il retransmet tous les paquets, même ceux qui ne sont pas perdus. Pendant la

congestion avoidance, la fenêtre de congestion est augmentée de $^{1}/_{cwnd}$ à la réception d'un nouvel ACK, alors qu'en Fast Recovery, elle est augmentée de 1.
Si ceci se produit à plusieurs reprises de manière successive, le bord gauche de la fenêtre d'émission évolue seulement après chaque Fast Retransmit successif et la quantité de données en vol (segments émis mais non acquittés) devient plus grande que la fenêtre de congestion cwnd (divisée en deux après le dernier appel de Fast Retransmit). Ainsi il n'y a plus d'accusés de réception à recevoir, alors le TCP émetteur bloque jusqu'à ce qu'un timeout se produise et déclenche un slow-start.

3.3 TCP NewReno

Il implémente TCP Reno avec une modification du Fast Recovery. Cette amélioration permet d'éliminer le défaut de l'algorithme de congestion en cas de pertes multiples au sein d'une même fenêtre.

Reno sort de la phase Fast Recovery lorsque le premier acquittement qui fait avancer la fenêtre arrive à l'émetteur. Cependant, s'il y a plus d'un segment perdu dans la même fenêtre, le Fast Retransmit n'est pas performant (voir section 3.2). Par conséquent, une alternative appelée NewReno a été suggérée par J. Hoe pour améliorer l'exécution de TCP.

Il n'y a pas de slow-start à la première congestion. A la réception d'un acquittement partiel, il retransmet immédiatement le premier paquet suivant le numéro de séquence de l'accusé de réception, considéré comme perdu. NewReno TCP quitte le Fast Recovery uniquement lorsque tous les segments dans la dernière fenêtre ont été acquittés avec succès, par la réception d'un fullACK.

Phases successives :

- slow-start
- congestion avoidance
- Fast Retransmit
- Fast Recovery amélioré

Cet algorithme est utilisé lorsque l'option des acquittements sélectifs, dont il est question ci-après, n'est pas employée.

3.4 Les acquittements sélectifs (SACK)

Le rétablissement des pertes de paquets est inefficace dans le TCP standard parce que les acquittements cumulatifs permettent seulement une retransmission à chaque RTT.

Les acquittements sélectifs (**selective acknowledgements**, SACK) est une stratégie qui corrige ce comportement face aux multiples pertes de segments [23]. Avec des acquittements sélectifs, le récepteur de données peut informer l'émetteur au sujet de tous les segments qui sont arrivés avec succès, ainsi l'émetteur retransmet seulement les segments qui ont été perdus réellement. Ceci permet à l'émetteur de faire plus d'une retransmission dans un RTT [29].

Plus précisément, la disponibilité de l'information SACK permet à l'émetteur d'effectuer le contrôle de congestion de manière plus fiable. Au lieu d'ajuster temporairement la fenêtre de congestion, l'émetteur peut estimer la quantité de données envoyées mais non acquittées (outstanding data) et la comparer à la fenêtre de congestion et décider s'il peut transmettre de nouveaux segments.
Cependant, les segments non acquittés peuvent être calculés de plusieurs manières. L'approche conservatrice est de considérer tous les segments non acquittés comme envoyés dans le réseau.

Le SACK peut être utilisé seulement si les extrémités de la connexion TCP le supportent.

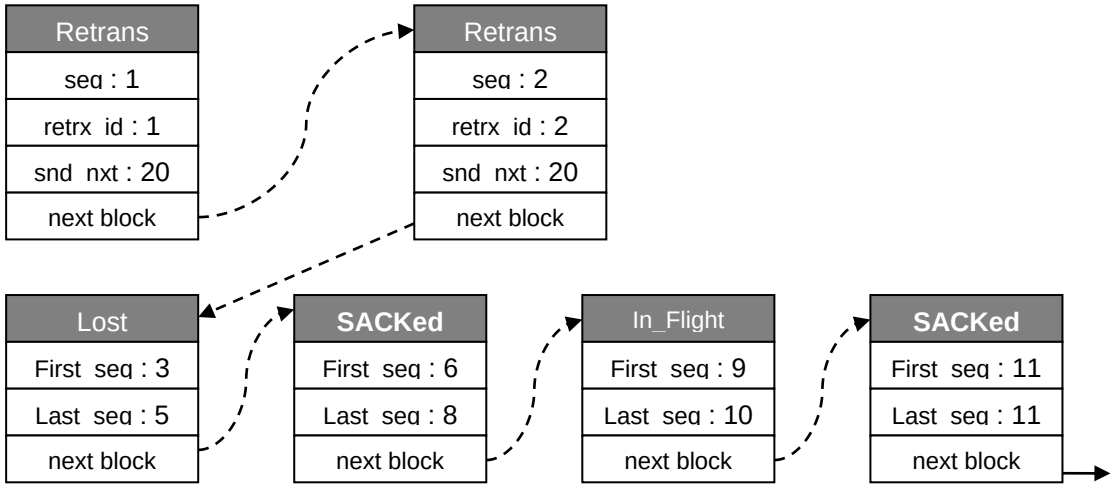


Figure 8 : SACK queue data structure [11]

Sur la figure 8, les listes des paquets qui ont été bien reçus par le récepteur sont en état SACKed.

3.5 L'algorithme Forward Acknowledgement (FACK)

L'algorithme **Forward Acknowledgement** (FACK) adopte une approche plus agressive et considère les trous non acquittés entre les blocs de SACK comme paquets perdus (figure 8 - Lost). Bien que cette approche ait souvent comme conséquence une meilleure performance

pour TCP que l'approche conservatrice, elle est carrément agressive si des paquets sont réordonnés dans le réseau, parce dans ce cas-ci les vides entre les blocs de SACK n'indiquent pas des paquets perdus.

3.6 Les Duplicate SACK (D-SACK)

Les blocs de SACK peuvent également être utilisés pour signaler de fausses retransmissions. Une amélioration, le Duplicate SACK (D-SACK) [24], permet au récepteur TCP de signaler tous les segments doubles qu'il obtient avec l'utilisation des blocs SACK.

Avec cette information l'émetteur TCP peut conclure dans certaines circonstances s'il a inutilement réduit ses paramètres de contrôle de congestion, et réinitialiser ainsi ces paramètres aux valeurs précédant la retransmission. Par exemple, le réordonnancement des paquets est une raison importante de retransmissions inutiles, parce que l'arrivée de segments en désordre déclenche l'envoi de dupACK par le récepteur.

3.7 Le Timestamp

L'option timestamp de TCP a été suggérée pour permettre des mesures plus précises des RTT, particulièrement sur les réseaux à produit délai-bande passante élevé. Un timestamp est associé à chaque segment, qui à l'acquittement du segment en fait écho. À partir de l'écho du timestamp l'émetteur peut mesurer des RTT précis pour les segments et employer la mesure pour éviter les RTO. En plus d'une mesure plus précise du RTT, l'utilisation des timestamp TCP permet aux algorithmes de rejeter des segments de connexions TCP antérieures qui traînent sur le réseau.

L'option timestamp permet également la détection des retransmissions inutiles. L'algorithme d'Eifel [27] suggère que si un acquittement pour un segment retransmis fait écho un timestamp plus tôt que le timestamp de la retransmission enregistré chez l'émetteur, alors le segment original était déjà parvenu au récepteur et la retransmission a été déclenchée inutilement. Dans ce cas, l'émetteur TCP peut continuer à envoyer de nouvelles données et rétablir ses paramètres de contrôle de la congestion à la situation antérieure à la retransmission.

3.8 L'Explicit Congestion Notification (ECN)

Au lieu d'induire la congestion des paquets perdus, Explicit Congestion Notification a été suggéré [25] pour que les routeurs marquent explicitement les paquets quand ils arrivent à un point encombré du réseau. Quand l'émetteur TCP reçoit un ECN du récepteur, il doit réduire

son débit de transmission pour atténuer la congestion dans le réseau. ECN permet aux émetteurs TCP d'être « congestion-aware » sans souffrir nécessairement des pertes de paquets.

3.9 TCP Vegas

Phases successives :

- slow-start modifié (cwnd doublée une fois sur deux) [17].
- congestion avoidance
- Fast Retransmit
- Fast Recovery

Cette amélioration de TCP Reno est proposée par Lawrence Brakmo [26]. L'algorithme de TCP Vegas peut interagir avec n'importe quel autre protocole TCP. Proactif, il réduit le débit avant qu'une perte n'apparaisse. Il évite la congestion en anticipant les pertes, grâce à une réduction du débit en fonction des variations du RTT. Il fait varier la fenêtre de congestion à partir d'une comparaison du taux de transmission attendu par rapport au taux de transmission en cours.

Par rapport à TCP Reno, Vegas offre un débit supérieur de 35% à 70%, tout en réduisant les retransmissions de 50 à 80% [1].

La taille de la fenêtre de congestion est ajustée de la manière suivante :

- En cas de congestion, pour chaque RTT, l'algorithme fait une estimation de la quantité de données stockées dans les files d'attente intermédiaires. Il décide alors s'il doit augmenter ou réduire la taille de la fenêtre de congestion.

- Il calcule le débit maximum :

  $$Expected = cwnd / BaseRTT$$

  où cwnd est la taille de la fenêtre et BaseRTT, la valeur minimum de tous les RTT mesurés.

- Vegas enregistre le nombre d'octets transmis d'un paquet particulier. Le temps d'aller- retour de ce paquet particulier est RTTsample. Vegas calcule le débit réel de transmission :

  $$Actual = cwnd / RTTsample$$

- Diff = Expected - Actual.

- L'émetteur définit 3 seuils : *alpha*, *beta* et *gamma*.

- Si Diff ≤ *alpha* / BaseRTT, alors pour chaque acquittement reçu, la fenêtre de congestion est augmentée de $^1/_{cwnd}$.

- Si Diff > *beta* / BaseRTT, la fenêtre de congestion est décrémentée d'un.

- Si Diff est comprise entre *alpha* et *beta*, la fenêtre reste inchangée.

Dans la version implémentée de TCP Vegas, RTTsample est substitué par RTTsmooth qui est la moyenne des RTT, calculée sur l'ensemble des paquets acquittés, entre le moment de la transmission du paquet particulier et la réception de son acquittement.

Un nouveau slow-start est introduit visant à minimiser les pertes durant cette phase :

- Vegas double la taille de la fenêtre un RTT sur deux.

- La phase du slow-start prend fin une fois que la fenêtre atteint une certaine valeur pour laquelle Diff > *gamma* / BaseRTT. Cette technique permet d'éviter des pertes pendant la phase de slow-start.

3.10 TCP Westwood

L'idée de Westwood [19] est d'estimer la bande passante disponible de façon plus précise que TCP, en se basant sur la vitesse de réception des messages d'acquittement. Cette estimation est alors utilisée pour le calcul du seuil du slow-start, ce qui permet à l'algorithme de perdre moins de temps à retrouver un régime normal après une congestion. Le protocole utilise également d'autres améliorations apportées à TCP, comme le Fast Recovery et le Fast Retransmit.

3.11 TCP BIC

L'objectif de TCP BIC [20] par rapport aux autres protocoles de transports pour réseaux longue distance et haut débit est d'assurer une équité entre des flux ne possédant pas le même RTT. Cela est réalisé par sa fonction d'évolution de la fenêtre de congestion, qui utilise une recherche binaire pour détecter une congestion : la taille augmente plus vite lorsque le seuil est éloigné, et ralentit lorsqu'il se rapproche. Cette recherche s'ajoute à l'algorithme

AIMD (avec toutefois des variables d'incrémentation plus élevées), ce qui permet à la fenêtre d'augmenter plus rapidement lorsqu'il reste beaucoup de bande passante disponible.

3.12 TCP Veno

Phases successives :

- slow-start
- congestion avoidance
- Fast Retransmit
- Fast Recovery

TCP Veno [1] est un algorithme de contrôle de congestion des connexions bout à bout qui peut améliorer de manière significative les performances du protocole TCP dans des réseaux hétérogènes, particulièrement des liaisons sans-fil.

L'innovation principale dans Veno est le perfectionnement de l'algorithme de contrôle de congestion Reno en utilisant une estimation de l'état de la connexion sur base de l'algorithme Vegas.
L'algorithme de Veno diminue de manière significative la réduction « à l'aveugle » de la fenêtre de congestion de Reno, indépendamment de la cause de la perte de paquet. Veno surveille le niveau de congestion du réseau et utilise cette information pour décider si la perte d'un paquet est due à une congestion sévère ou à une perte aléatoire de bit (bit error).

Plus précisément, Veno améliore l'algorithme de Multiplicative Decrease de Reno en affectant une valeur au seuil de slow-start en fonction de la qualité estimée de la connexion plutôt qu'à un coefficient fixe. Ensuite il optimise l'algorithme de Additive Increase pour qu'il utilise toute la largeur de bande disponible du réseau.

Toutes les autres parties de Reno, slow-start initial, Fast Retransmit et Fast Recovery ne sont pas modifiées.

Un autre avantage de TCP Veno est qu'il n'y a pas besoin de modification du côté du récepteur.

Dans Vegas (voir section 3.9), l'émetteur mesure la différence Diff entre les taux Expected et Actual.

Quand RTTsmooth > BaseRTT, il y a signe qu'il y a une connexion qui ralentit et que des paquets sont accumulés. Si N est le nombre des paquets accumulés dans la pile d'attente du lien qui ralentit, alors

RTTsmooth = BaseRTT + N / Actual

Si les délais supplémentaires au goulot d'étranglement sont attribués dans la seconde partie de la formule ci-dessus, en réarrangeant on obtient,

N = Actual * (RTTsmooth – BaseRTT) = Diff * BaseRTT

L'idée principale de Veno, contrairement à Vegas qui utilise N pour ajuster sa fenêtre de congestion de manière proactive, est que N peut être utilisé comme indication pour définir si le réseau est dans une situation de congestion sévère ou pas.

L'idée essentielle de Reno selon laquelle la fenêtre de congestion évolue progressivement pendant la non-détection de perte, reste intacte aussi à Veno.

Néanmoins, si N < *beta* au moment de la détection d'une perte, Veno considère qu'il ne s'agit pas d'une congestion sévère et que la fenêtre de congestion doit évoluer d'une manière différente à celle de Reno. Sinon, Veno considère qu'il s'agit d'une congestion sévère et il adopte le mécanisme de Reno pour l'incrémentation de la fenêtre de congestion. Par des expériences *beta* a été défini à 3.

Il faut rappeler que dans l'algorithme original de Vegas, BaseRTT est continuellement mis à jour tout au long de la connexion TCP avec le plus petit des RTT mesurés.

Dans Veno, cependant, BaseRTT est initialisé chaque fois qu'une perte de paquet est détectée, à cause d'un timeout ou d'un dupACK. BaseRTT est alors mis à jour comme dans l'algorithme original de Vegas jusqu'au démarrage du prochain Fast Recovery ou slow-start.

Ceci est fait pour tenir compte du trafic changeant des autres connexions. La largeur de bande acquise par une connexion simple, parmi les nombreuses connexions, peut changer de temps en temps, faisant changer également le BaseRTT.

Slow-start

Veno utilise l'algorithme slow-start de Reno.

Congestion Avoidance

Dans Veno l'algorithme Additive Increase de Reno est modifié comme suit [1] :

```
if ( N < beta )                    // available bandwidth not fully utilized
        set cwnd = cwnd + 1 / cwnd   when each new ACK is received
else
```

if (N ≥ *beta*) // available bandwidth fully utilized
set cwnd = cwnd + 1 / cwnd when every other new ACK is received

La première partie est identique à Reno. Dans la seconde partie, si le nombre des paquets accumulés dans le buffer *N* dépasse *beta*, alors la fenêtre de congestion est incrémentée de 1 à tous les 2 RTT, ainsi, la connexion reste plus longtemps dans une zone opérationnelle.

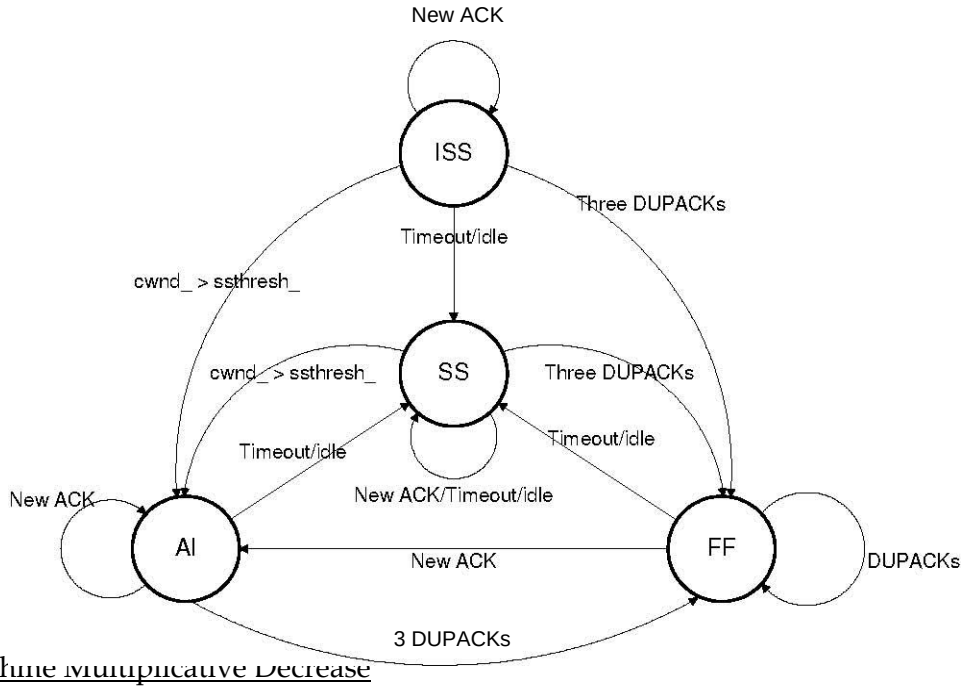


Figure 9 : évolution des états d'une connexion TCP Veno [20]

L'algorithme Multiplicative Decrease

En Reno, il y a deux moyens pour détecter la perte d'un paquet.

Un paquet est considéré comme perdu quand il n'a pas encore été acquitté par le récepteur à l'expiration du temporisateur. Dans ce cas l'algorithme de slow-start est initialisé avec le ssthresh = ½ cwnd et cwnd = 1. Ce comportement a comme effet de réduire le débit, parce que l'expiration du timer est considérée comme signe d'une congestion sévère. Veno ne modifie pas cette partie de l'algorithme.

Dans le cas de la réception de 3 dupACK, Reno utilise le mécanisme Fast Retransmit - Fast Recovery, comme décrit à la section 2.2.2. Veno modifie la première partie de l'algorithme [1]:

```
if (N < beta)     // random loss due to bit errors is most likely to have
                                                                  occurred
      ssthresh = cwnd * (4/5) ;
else                    // congestive loss is most likely to have occurred
      ssthresh = cwnd/2 ;
```

Ainsi, selon Veno, si la connexion n'est pas en congestion sévère (*N* < *beta*) alors il s'agit d'une perte aléatoire de bit (random bit error). Veno réduit la taille de la fenêtre de

congestion et du seuil ssthresh d'un cinquième au lieu de la moitié de la taille atteinte au moment de la détection de la perte.

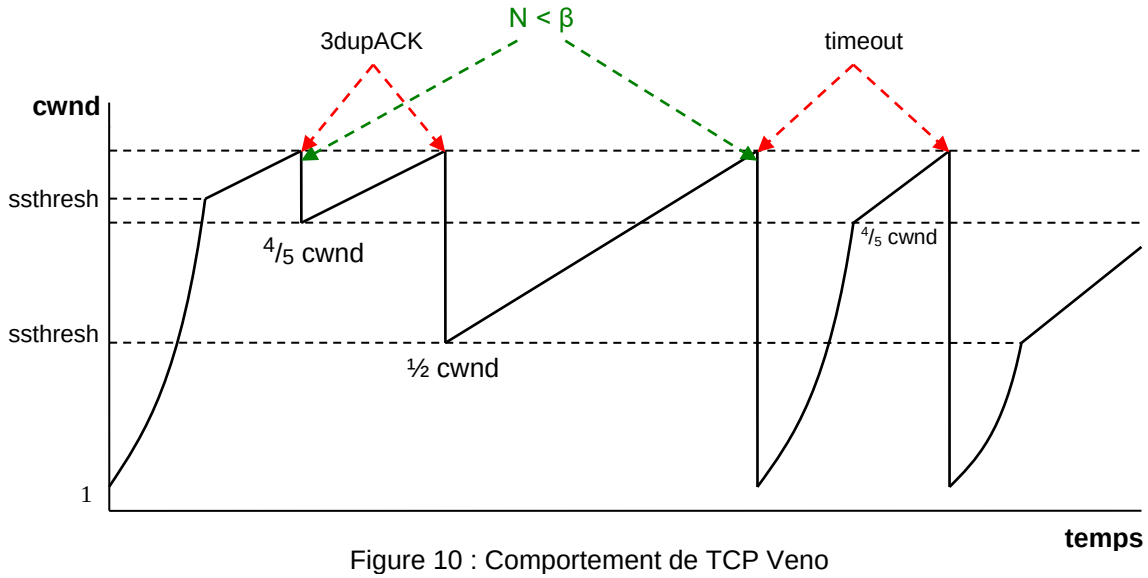


Figure 10 : Comportement de TCP Veno

La variable beta (β), TCP Veno vs TCP Vegas

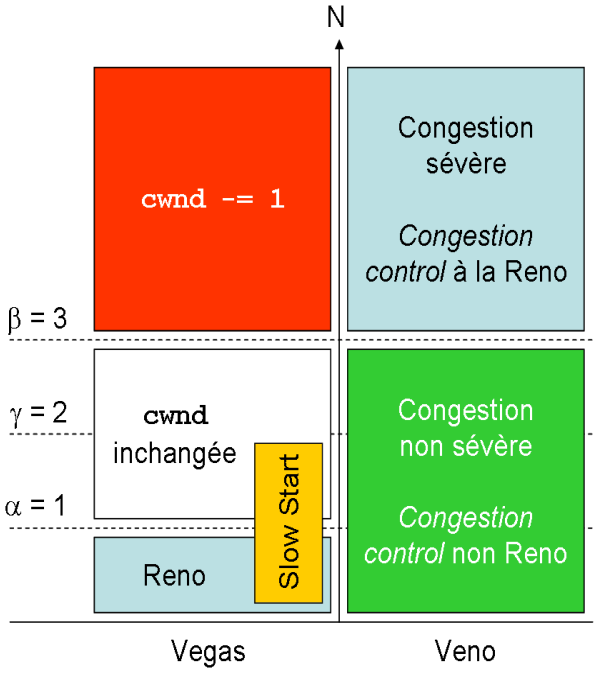


Figure 11 : Vegas vs Veno [30]

- N = Diff * BaseRTT
- Diff est le nombre estimé des paquets sur le réseau
- BaseRTT varie de la même manière pour les deux connexions.

Dans Vegas, tant que Diff est situé dans la zone optimale entre α et β, le flux de données est maintenu (figure 11) [30].

Dans Veno, au contraire, β reste fixe à 3, (valeur expérimentale pour définir une limite optimale). Le flux de données varie selon la taille de la fenêtre de congestion.

3.13 GTCP

GTCP [2] peut être utilisé dans n'importe quelle architecture qui supporte des réservations de largeur de bande. Il est prévu pour des architectures réseaux avec QoS pour applications utilisant des largeurs de bande réservées qui exigent du protocole de transport des services

de livraison garantis, ce que le service best-effort de TCP est par nature incapable d'offrir. En effet TCP est conçu à la base pour un environnement où tous les flux sont des flux best-effort sans assurance de largeur de bande [2].

GTCP offre aux applications exigeantes les débits souhaités. Il ne diffère de TCP que par son mécanisme de contrôle de la congestion. Il adapte le slow-start, la congestion avoidance et les mécanismes de Fast Recovery et de timeout de TCP pour réaliser des meilleures performances.

Dans une situation de trafic de réseau avec largeur de bande garantie, la fenêtre de congestion a deux composantes logiques qui correspondent aux services offerts. Une composante (service) garanti et une composante best-effort : cwnd = $cwnd_G$ + $cwnd_{BE}$. La composante $cwnd_{BE}$ est traitée par GTCP de la même manière que si la cwnd n'avait pas de composante service garanti, néanmoins cwnd est une seule entité.

En TCP, cwnd = rate * BaseRTT,

en GTCP, cwnd = ($rate_G$ + $rate_{BE}$) * BaseRTT.

où $rate_G$ constitue l'engagement de service garanti et BaseRTT est le plus petit des RTT enregistrés pendant la connexion,

BaseRTT = min (BaseRTT, RTTsample)

Le Self-clocking

Le comportement idéal pour une connexion avec réservation de largeur de bande, doit être celui où l'algorithme transmet au moins

$$cwnd_G = rate_G * BaseRTT$$

segments pour chaque RTT. Ce comportement est maintenu tant qu'il n'y a pas de perte. Cependant, quand des pertes se produisent, le bord gauche de la fenêtre de congestion ne peut pas être décalé vers la droite, en attendant l'arrivée d'un ACK pour ce premier numéro de séquence non acquitté.

Le comportement de démarrage slow-start

- cwnd est initialisée à 2 (parce que 2 composantes).
- Pour chaque ACK reçu : cwnd++.
- GTCP quitte le slow-start quand cwnd > $cwnd_G$ + ssthresh.

Congestion avoidance :

- GTCP utilise le même procédé de congestion avoidance que TCP :

$$cwnd = cwnd + {}^{1}/_{cwnd}.$$

- En cas de perte détectée, si on considère k le nombre de pertes pendant la période de la récupération de la perte, alors le nombre idéal de nouveaux segments qui doivent être envoyés à chaque RTT doit être :

$$FRideal = \max (cwnd_G, cwnd - k - \tfrac{1}{2}\, cwnd_{BE})$$

- Le Fast Recovery de TCP NewReno, n'aurait pas assuré le self-clocking en cas de pertes importantes.

Si le nombre de pertes k est plus grand que la moitié du composant best-effort de la fenêtre de congestion, le TCP n'enverra pas la quantité désirée de segments. Afin de résoudre ce problème, GTCP emploie un mécanisme modifié de Fast Recovery qui est décrit ci-dessous :

Le mécanisme modifié de Fast Recovery :

1) $cwnd_{update} = cwnd_G + \tfrac{1}{2}\, cwnd_{BE}$, avant l'initialisation de cwnd à ssthresh.

2) les premiers $cwnd_G$ dupACK, juste après la détection de perte, provoquent la transmission d'autant de nouveaux segments et chaque fois la *cwnd* est incrémentée.

3) Ensuite les $\tfrac{1}{2}\, cwnd_{BE}$ dupACK (ou les $cwnd_{BE} - k$, si $k > \tfrac{1}{2}\, cwnd_{BE}$) suivants sont ignorés (aucune transmission n'est déclenchée et la cwnd demeure inchangée !).

4) GTCP transmet de nouveaux segments de données pour les dupACK suivants.

5) Quand un fullACK arrive, GTCP initialise sa fenêtre cwnd à $cwnd_{update}$.

Quand le nombre de pertes $k < \tfrac{1}{2}\, cwnd_{BE}$, le comportement de GTCP est exactement identique à celui de TCP NewReno, mais l'ordre dans lequel la fenêtre est augmentée et ensuite $cwnd_G$ nouveaux paquets de données est transmis, est commuté.
Puisqu'il est assuré que $cwnd_G$ paquets sont sûrement transmis par le réseau, il est certain qu'il y a assez de dupACK qui reviennent pour déclencher au moins $cwnd_G$ paquets supplémentaires après l'indication de la perte.

Néanmoins, quand k > ½ $cwnd_{BE}$, au moins $cwnd_G$ nouveaux segments sont envoyés sur le réseau, ce qui correspond au débit minimum garanti par le réseau.

Le mécanisme modifié de timeout recovery :

Par conséquent, GTCP est adapté pour maintenir le self-clocking pendant le timeout recovery de telle manière qu'au moins $cwnd_G$ segments soient transmis à chaque RTT. La conception est semblable à celle de l'algorithme modifié de Fast Recovery :

1) $cwnd_{BE}$ = ½ $cwnd_{BE}$ et $ssexit_{thresh}$ = maximum segment number.
   En ce qui concerne l'initialisation de la fenêtre de congestion cwnd, l'auteur de GTCP, est ambigu. Dans son document [2] il indique que cwnd=1 sans autre précision, alors que pour le démarrage de slow-start il annonce une initialisation à 2 (1 pour chaque composante).
   « When timeout occurs, the congestion window *cwnd* is collapsed to one (not $cwnd_G$+1), representing the true best effort component of the congestion window, …. »

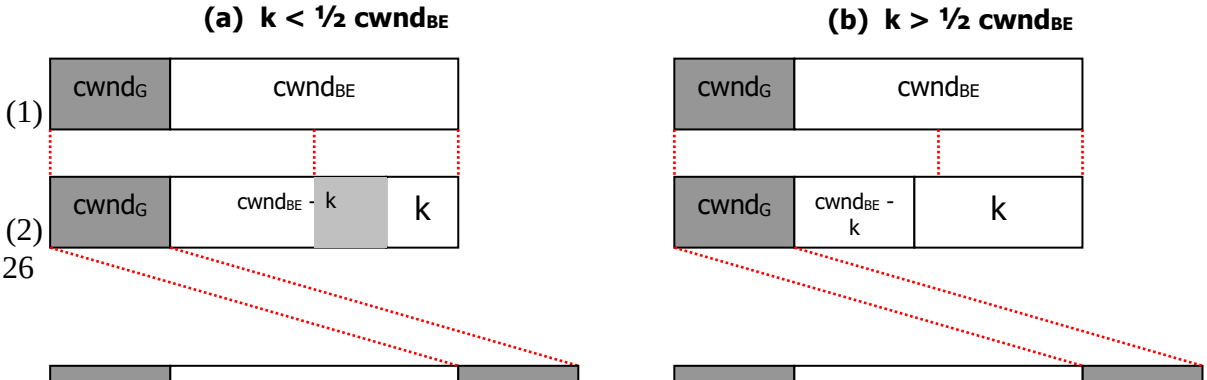
2) les premiers $cwnd_G$ dupACK, juste après le timeout, provoquent la transmission d'autant de nouveaux segments et chaque fois la cwnd est incrémentée.

3) Ensuite les (outstanding packets timeout - $cwnd_G$) dupACK suivants sont ignorés. Par outstanding packets timeout, on définit le nombre des paquets émis sur le réseau mais non encore acquittés par le récepteur au moment du timeout.

4) Au premier ACK reçu GTCP retransmet tous les segments restants dans la cwnd de la même manière que le slow-start de TCP.

5) Après la réception d'un ACK pour le paquet $ssexit_{thresh}$, alors GTCP quitte le slow-start et initialise la cwnd à $cwnd_G$ + $cwnd_{BE}$.

Notez que pour le timeout recovery, le GTCP exécute un slow-start seulement pour le composant best-effort de la cwnd et la largeur de bande réservée est maintenue au niveau de réservation. Par conséquent, le comportement de démarrage de GTCP est différent de celui du comportement de son timeout recovery.
La figure 12 ci-après offre une représentation graphique du fonctionnement de GTCP [2].



Explication : comme défini dans le RFC 3782 [7] à la sortie de Fast Recovery, cwnd est initialisée selon la formule définie dans la spécification :

```
cwnd = ssthresh = max(FlightSize / 2, 2 * SMSS)
```

où FlightSize est le nombre des paquets transmis mais non encore acquittés AU MOMENT de la sortie de la phase Fast Recovery. Si le nombre des paquets restants dans la nouvelle fenêtre est inférieur au nombre de paquets défini par la réservation de la largeur de bande, alors la garantie du débit fourni n'est pas respectée.

Chapitre 4

TCP Xeno

Ce chapitre aborde le sujet principal de ce travail, à savoir l'étude des possibilités d'interaction entre les protocoles GTCP et TCP Veno.

4.1 Approche théorique du problème.

En GTCP, la fenêtre de congestion est constituée de deux composantes, une composante pour le service garanti, $cwnd_G$, et l'autre, $cwnd_{BE}$, pour le service best-effort. Ainsi :

$$cwnd = cwnd_G + cwnd_{BE}$$

En cas de détection d'une perte par la réception de 3 dupACK, le débit minimum que l'algorithme doit garantir est le débit réservé par l'application. L'algorithme Multiplicative Decrease est appliqué uniquement sur la partie best-effort de la fenêtre de congestion.

Par conséquence, le nombre de nouveaux segments qui doivent être envoyés à chaque RTT (round-trip-time) est calculé selon la formule [2] :

$$FRideal = \max (cwnd_G, cwnd - k - \tfrac{1}{2} cwnd_{BE})$$

où cwnd - k est le nombre de segments envoyés sur le réseau mais non acquittés après k pertes.

En cas de pertes limitées, quelque soit la nature de la congestion qui a provoqué la perte, GTCP réduit $cwnd_{BE}$ de moitié de manière fixe. De la même manière en cas de timeout, la valeur du seuil de passage à la congestion avoidance sera $ssthresh = cwnd_G + \tfrac{1}{2} cwnd_{BE}$.

En TCP Veno, BaseRTT se calcule de la même façon que dans TCP Vegas, à la seule différence que BaseRTT est initialisé chaque fois qu'une perte de paquet est détectée, à la suite d'un timeout ou de la réception de 3 dupACK successifs. Ainsi,

$$BaseRTT = \min (BaseRTT, RTTsample).$$

et

$$N = Diff * BaseRTT$$

L'algorithme de TCP Veno effectue une estimation de l'état du réseau. Si $N < beta$ au moment de la détection d'une perte, Veno considère qu'il ne s'agit pas d'une congestion sévère et que la fenêtre de congestion doit évoluer de manière différente que le Multiplicative Decrease de TCP Reno. Ainsi, en cas de congestion légère le ssthresh est fixé à ⅘ cwnd.

Comparant le comportement de GTCP à celui de TCP Veno on constate que dans le cas d'une congestion légère, l'algorithme de GTCP réduit inutilement le débit de moitié, ce qui diminue de manière significative la performance du réseau.

De la même façon, après un timeout, GTCP atteint la bande passante disponible moins vite que TCP Veno, puisqu'il passe en congestion avoidance bien avant TCP Veno.

Il serait alors intéressant d'étudier la possibilité de compensation de cette faiblesse de GTCP en intégrant l'algorithme d'estimation de l'état du réseau de TCP Veno dans le mécanisme de contrôle de congestion de GTCP, pour donner naissance à TCP Xeno.

La figure 13 représente le comportement estimé de TCP Xeno. Il s'agit de l'association des comportements de GTCP (figure 14) et de TCP Veno (figure 10).

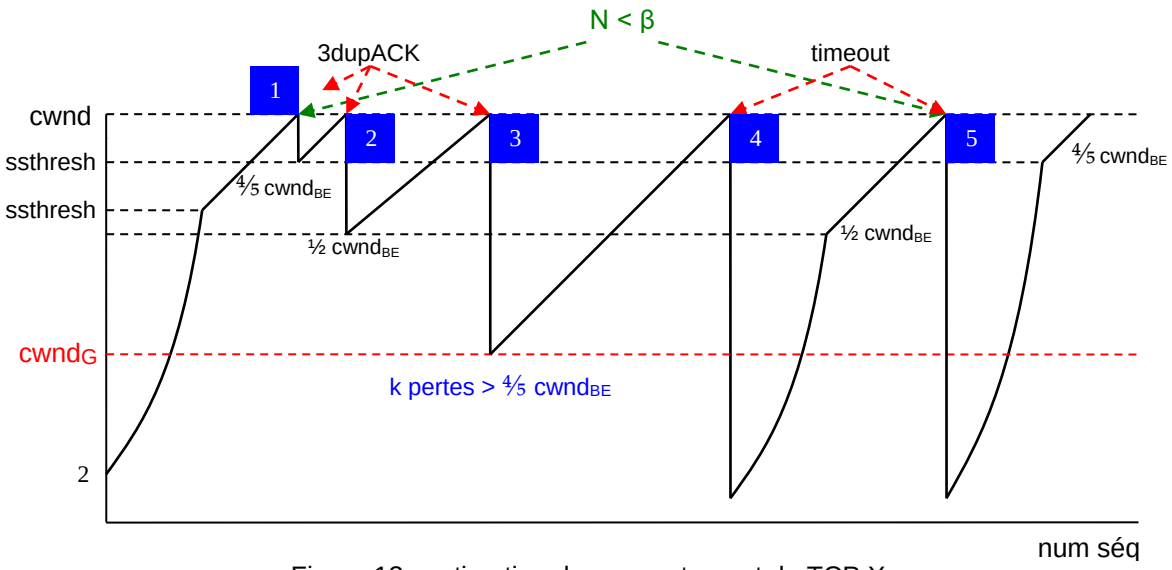


Figure 13 : estimation du comportement de TCP Xeno

Il y a 5 scénarios de perte possibles pour TCP Xeno (figure 12) :

1) 3dupACK, $N < \beta$, $k < \frac{4}{5}\ cwnd_{BE}$
2) 3dupACK, $N \geq \beta$, $k < \frac{4}{5}\ cwnd_{BE}$
3) 3dupACK, $k > \frac{4}{5}\ cwnd_{BE}$
4) timeout, $N \geq \beta$
5) timeout, $N < \beta$

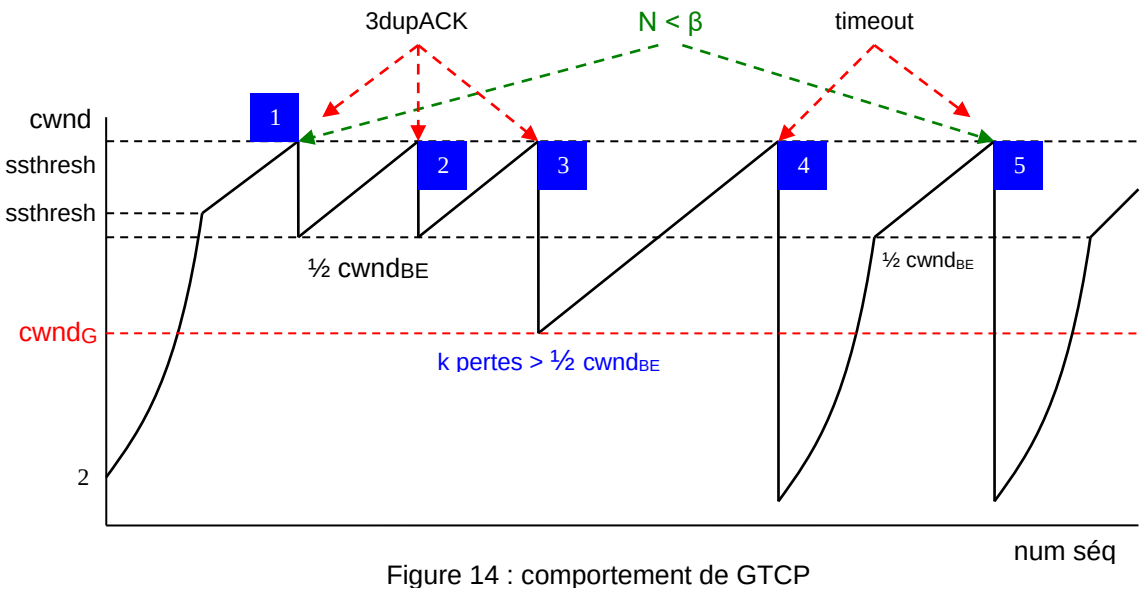


Figure 14 : comportement de GTCP

Les 5 scénarios sont applicables aussi dans le cas de GTCP. La figure 14 ci-dessus représente les 5 scénarios de la figure 13 adaptés à l'algorithme GTCP, c'est-à-dire que k est comparé à ½ cwndBE.

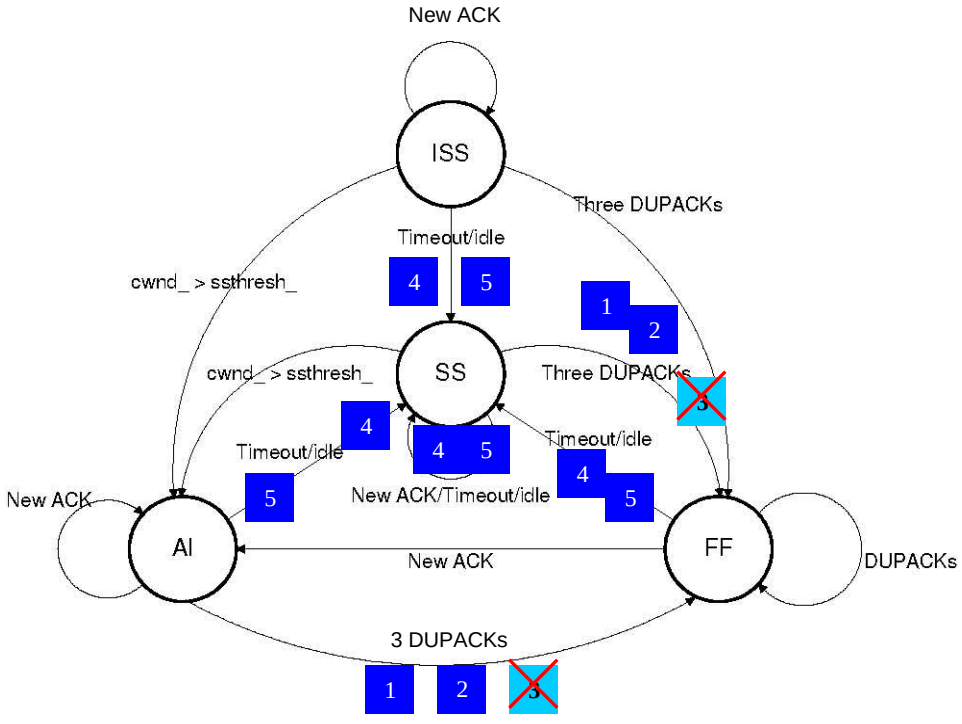


Figure 15 : Les 5 scénarios de perte dans une connexion TCP Veno

La figure 15 représente les scénarios de perte précédemment définis tels qu'ils peuvent apparaître dans une connexion TCP Veno. L'algorithme de Veno ne vérifie pas k. Ainsi, le scénario 3 n'a pas de sens d'y figurer.

En observant les représentations graphiques des comportements de TCP Xeno et de GTCP (figures 13 et 14), on constate que, théoriquement au moins, le premier atteint le débit maximal offert par le réseau beaucoup plus rapidement que le second.

4.2 Structure de l'algorithme proposé

4.2.1 Estimation du RTT et de la fenêtre de congestion

- En GTCP

$$cwnd = cwnd_G + cwnd_{BE}$$
$$cwnd = (rate_G + rate_{BE}) * BaseRTT$$
$$BaseRTT = \min (BaseRTT, RTTsample).$$

où $rate_G$ est le débit réservé par l'application que l'algorithme doit garantir, $cwnd_G$ est le composant largeur de bande garantie de la fenêtre de congestion cwnd, $cwnd_{BE}$ est sa partie best-effort et BaseRTT est le plus petit RTT pour la connexion.

- En TCP Veno

$$cwnd = rate_{BE} * BaseRTT$$

où BaseRTT est le RTT minimum mesuré depuis son initialisation, après la détection d'une perte. Donc,

$$BaseRTT = \min (BaseRTT, RTTsample).$$

Comme indiqué à la section 3.12, l'amélioration que TCP Veno apporte à TCP Reno est située dans la phase de congestion avoidance de l'algorithme de contrôle de congestion.

Pour rappel, dans TCP Veno, cwnd est incrémentée de la manière suivante :

si $N < beta$,
alors cwnd est incrémentée à la réception de chaque nouvel ACK reçu
sinon
cwnd est incrémentée à la réception d'un sur deux nouvel ACK reçu.

En cas de détection de perte, suivant l'état du réseau :

Si pas de congestion ($N < beta$),
$cwnd = cwnd_G + \frac{4}{5}\, cwnd_{BE}$ (a)
sinon, congestion sévère,
$cwnd = cwnd_G + \frac{1}{2}\, cwnd_{BE}$ (b)

En cas de congestion, s'il y a *k* pertes et que cwnd est la taille de la fenêtre de congestion au moment de la détection de la perte, alors le nombre idéal de segments qui doivent être envoyées par RTT dépend de l'état du réseau [1].

Ainsi, en cas de congestion, le calcul du nombre idéal FRideal est celui fourni par GTCP [2].

D'autre part en tenant compte de (a) en cas de non congestion le nombre idéal de segments envoyés doit être :

$$\mathbf{FRideal_{NC} = max\ (cwnd_G,\ cwnd - k - \tfrac{1}{5}\ cwnd_{BE})}$$

Ainsi en cas de congestion (b) le nombre de segments envoyés est le même qu'en GTCP :

$$FRideal_C = max\ (cwnd_G,\ cwnd - k - \tfrac{1}{2}\ cwnd_{BE})$$

En utilisant le même raisonnement que [2], si le nombre de pertes *k* est supérieur à $\frac{4}{5}$ $cwnd_{BE}$, TCP ne peut plus offrir la largeur de bande souhaitée, de même que pour GTCP (voir fig. 1 dans [2]).

Il faut remarquer pour le cas de pertes $k > \frac{4}{5}\ cwnd_{BE}$, l'algorithme modifié de GTCP est toujours applicable puisqu'il gère les cas où le nombre de pertes $k > \frac{1}{2}\ cwnd_{BE}$. Néanmoins, ce comportement ne satisfait pas $FRideal_{NC}$, donc il n'est pas optimal.

4.2.2 L'algorithme de TCP Xeno

Slow-start

Xeno aura le même comportement de démarrage que GTCP. Le slow-start est le même pour $cwnd_G$ et pour $cwnd_{BE}$. Au démarrage *cwnd* doit être initialisé à 2. Pour chaque ACK reçu cwnd est incrémenté de 1. Xeno sort du slow-start lorsque cwnd > sshtresh + $cwnd_G$ [2]

Congestion Avoidance

Xeno exécute l'algorithme Additive Increase de la même manière que Veno [1] :

Si N < *beta*,
    cwnd += $1/cwnd$; à chaque RTT
sinon
    cwnd += $1/cwnd$; tous les deux RTT

Comme pour GTCP, Xeno ne maintient pas les composants garanti et best-effort séparés, cwnd est la somme de $cwnd_G$ et de $cwnd_{BE}$.

Fast Recovery

Le Fast Recovery est important pour les connexions avec largeur de bande réservée sans quoi de nouvelles données ne peuvent pas être envoyées pendant la période de récupération de la perte :

1) Quand une perte est détectée, avant que la fenêtre de congestion de soit mise à jour, elle est enregistrée dans $cwnd_{update}$:

   si congestion sévère alors
   $cwnd_{update} = cwnd_G + ½\ cwnd_{BE}$
   sinon $N < \beta$
   $cwnd_{update} = cwnd_G + ⅘\ cwnd_{BE}$

2) Xeno effectue des transmissions pour les $cwnd_G$ premiers dupACK qui suivent immédiatement la perte détectée et « gonfle » la fenêtre de congestion à chaque transmission.

3) Après $cwnd_G$ transmissions Xeno ignore les :

   si $N< \beta$: ⅕ $cwnd_{BE}$ dupACK suivants (ou les $cwnd_{BE}$ - k, si k > ⅘ $cwnd_{BE}$),
   sinon : ½ $cwnd_{BE}$ dupACK suivants (ou les $cwnd_{BE}$ - k, si k > ½ $cwnd_{BE}$).

   Ainsi cwnd n'évolue pas et il n'y a pas de transmission de segments.

4) Xeno transmet de nouveaux segments pour les dupACK suivants, s'il y en a.

5) A l'arrivée d'un FullACK, Xeno initialise la cwnd à $cwnd_{update}$.

Cet algorithme est pratiquement identique à celui décrit dans [2]. Seules les valeurs numériques changent. Ainsi tout comme GTCP, l'algorithme de Fast Recovery de Xeno satisfait le $FRideal_{NC}$.

Timeout Recovery

Le comportement de Xeno est exactement le même que celui de GTCP. Pendant le timeout recovery, Xeno adapte son self-clocking de manière à transmettre au moins $cwnd_G$ segments à chaque RTT. Sa conception est similaire à celle du mécanisme de Fast Recovery modifié, expliqué précédemment :

1) A cause du problème d'ambiguïté, concernant l'initialisation de la cwnd, expliqué à la section 3.13, il a été décidé qu'au timeout, cwnd est initialisée à 1, de la même manière qu'en TCP. $ssexit_{thresh}$ = maximum sequence number.

2) Pour les $cwnd_G$ premiers dupACK, à chaque dupACK reçu, Xeno transmet un nouveau segment.

3) Les dupACK suivants sont ignorés, jusqu'à ce que le nombre total des dupACK reçus dépasse le nombre des paquets envoyés mais non acquittés (outstanding packets) au moment du timeout.

4) Ensuite, pour chaque dupACK arrivé, un nouveau segment est transmis.

5) A l'arrivée d'un ACK, l'algorithme adopte le même comportement que le slow-start de TCP.

6) Pour terminer, quand le bord gauche de la fenêtre offerte passe au-delà de $ssexit_{thresh}$, Xeno sort du slow-start et $cwnd = cwnd_G + cwnd_{BE}$.

Chapitre 5

L'implémentation

5.1 Introduction

Le lecteur déjà familier avec le kernel peut passer ce chapitre.

Une fois l'approche théorique de l'algorithme établie, il reste la démonstration expérimentale des avantages que, le nouveau protocole est supposé apporter à la gestion de contrôle de la congestion, de GTCP.

Il faut alors choisir l'environnement dans lequel le prototype du protocole sera développé et testé. Pour cela, il faut un système ouvert qui permet l'intégration, si possible aisée, du nouveau protocole dans la gestion de sa couche de transport.

Le seul système connu qui réponde à ce besoin, est Linux.

Linux est un système d'exploitation de type Unix-like, dont la popularité a augmenté très fortement ces dernières années. Le choix de ce système offre deux avantages très importants pour l'implémentation du nouveau protocole :

- Le code-source du cœur du système Linux, le **kernel**, est publié [21] et son utilisation est totalement libre. De plus, une documentation très riche concernant sa structure et les modules qui le constituent est disponible sur Internet et sur les forums de discussions des développeurs.

- L'autre avantage, non négligeable, est que plusieurs des algorithmes de gestion de la congestion ont déjà été développés pour ce système. Ils constituent une mine d'informations pour la constitution du nouveau module.

Ce chapitre décrit la structure des principales fonctionnalités Linux, se concentrant sur les algorithmes de commande de la congestion. Au fur et à mesure de la présentation les différences avec le TCP traditionnel seront expliquées. En conclusion, la possibilité d'y intégrer le TCP Xeno comme module sera discutée.

5.2 Linux et TCP

Linux a intégré plusieurs de spécifications RFC dans un mécanisme de contrôle de la congestion, lequel supporte tant le TCP SACK que TCP NewReno sans information SACK. En outre, Linux a amélioré certaines des ces spécifications afin d'augmenter l'efficacité du mécanisme TCP. Ainsi, certains détails où le comportement de Linux TCP diffère des réalisations conventionnelles de TCP ou des spécifications de RFC seront précisés.

5.3 Linux et le contrôle de congestion

5.3.1 Estimation du nombre des paquets « envoyés sur le réseau et non acquittés »

Linux emploie l'ensemble de concepts et de fonctions du recovery de NewReno et de deux éléments de SACK pour déterminer le nombre de paquets « envoyés sur le réseau mais non acquittés » (outstanding packets). Quand l'information de SACK peut être utilisée, l'émetteur peut soit suivre les Forward Acknowledgements (FACK) et considérer les trous entre les blocs de SACK comme des segments perdus, soit dans une approche plus conservatrice, considérer tous les segments non acquittés comme « outstanding » sur le réseau. Dans tous les mécanismes de recovery, Linux utilise les équations suivantes :

```
left_out = sacked_out + lost_out

in_flight = packets_out - left_out + retrans_out
```

pour définir le nombre des paquets « outstanding » sur le réseau [21].

Dans l'équation ci-dessus,

- `packets_out` : nombre de segments déjà transmis situés au-dessus de `snd_una`,
- `sacked_out` : nombre de segments acquittés par des blocs de SACK,
- `lost_out` : une estimation du nombre de segments perdus dans le réseau, et
- `retrans_out` : nombre de segments retransmis.

La détermination du paramètre `lost_out` dépend de la méthode de recovery choisie. Par exemple, quand FACK est en service, tous les segments non acquittés entre le bloc de SACK le plus élevé et le fullACK sont comptés dans `lost_out`. La méthode choisie rend facile l'ajout de nouvelles heuristiques pour estimer les segments perdus.

En l'absence d'information de SACK, TCP incrémente sacked_out de 1 à chaque dupACK reçu. Ceci est en conformité avec les spécifications de contrôle de congestion de TCP, et le comportement qui en résulte est semblable à celui de l'algorithme de NewReno avec ses transmissions forcées. Linux a choisi de ne pas exiger d'adaptation arbitraire de la fenêtre de congestion, mais la taille de cwnd est le nombre de segments considérés « outstanding » sur le réseau pendant le Fast Recovery.

5.3.2 Le tableau indicateur scoreboard

Les compteurs utilisés pour estimer le nombre de paquets « outstanding », acquittés, perdus, ou retransmis exigent les structures de données additionnelles pour les supporter. TCP maintient l'état de chaque segment « outstanding » dans un tableau indicateur, le **scoreboard**, où il marque l'état connu du segment. Le segment peut être marqué comme « outstanding », acquitté, retransmis, ou perdu.

Des combinaisons de bits sont également possibles. Par exemple, un segment peut être déclaré perdu et retransmis, dans ce cas l'émetteur compte obtenir un acquittement pour la retransmission. En utilisant cette information Linux sait quels segments doivent être retransmis, et comment ajuster les compteurs utilisés pour calculer `in_flight` quand un acquittement nouveau arrive. Le scoreboard joue également un rôle important en déterminant si un segment a été considéré à tort comme perdu, par exemple en raison d'un réordonnancement de paquets.

5.3.3 Estimation des paquets perdus

Les inscriptions du scoreboard et les compteurs utilisés pour déterminer `in_flight` devraient être dans un état conforme à tout moment. Définir les segments comme « outstanding », acquittés et retransmis est relativement simple, mais pour qualifier un segment comme perdu, cela dépend de la méthode de recovery utilisée. Avec le recovery de NewReno, le premier paquet non acquitté est qualifié de perdu quand l'émetteur entre en phase Fast Recovery. En effet, ceci correspond aux spécifications RFC 2581[5] pour le contrôle de congestion. En outre, quand un ACK partiel arrive au début du Fast Recovery, le premier segment non reconnu est marqué perdu. Ceci a comme conséquence la retransmission du prochain segment non acquitté, comme l'exigent les spécifications de NewReno.

Quand SACK est utilisé, plusieurs segments peuvent être marqués comme perdus. Avec l'approche conservatrice, TCP ne compte pas les trous entre les blocs acquittés dans `lost_out`, mais quand FACK est actif, l'émetteur marque les trous entre les blocs de SACK perdus aussitôt qu'ils apparaissent. Le compteur `lost_out` s'ajuste de manière appropriée.

Chaque paquet dans la liste de retransmission de SACK (voir section 3.4) peut avoir 4 états possibles (figure 16) :

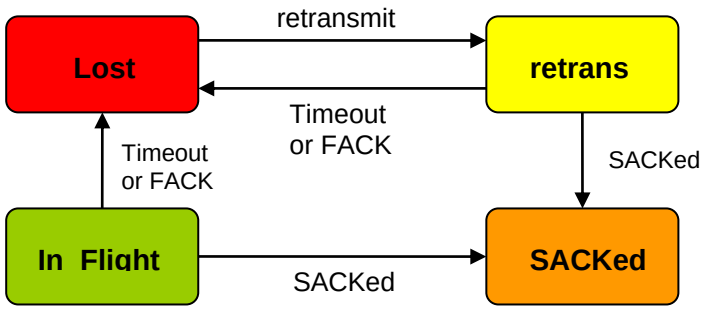


Figure 16: state machine of each packet [11]

5.3.4 Les états de contrôle de la congestion

Le TCP de Linux est régi par une machine d'états, la **Linux ACK finite machine**, qui détermine les actions de l'émetteur quand des acquittements arrivent. Le schéma 17 ci-dessus illustre comment les états sont reliés entre eux. Les états de contrôle de la congestion se présentent comme suit [4] [14]:

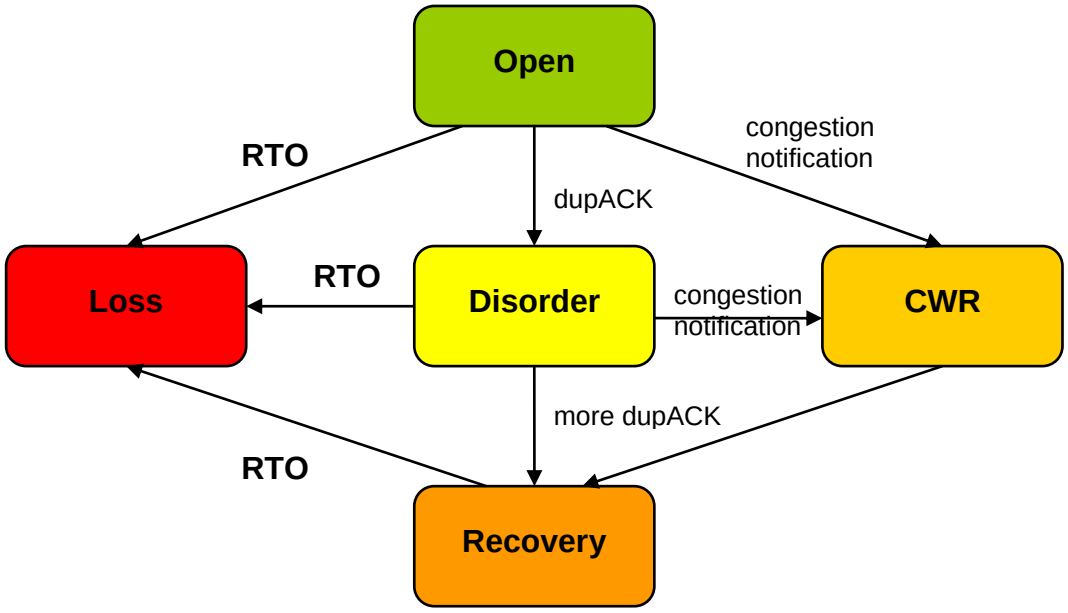


Figure 17: congestion control state machine [4]

Open : C'est l'état normal pendant lequel l'émetteur gère de manière optimale et rapide les cas des acquittements (ACK) entrants. Quand un acquittement arrive, l'émetteur incrémente sa fenêtre de congestion, selon que la fenêtre de

congestion est inférieure ou supérieure au seuil ssthresh, en phase slow-start ou en congestion avoidance. En Linux cet état est défini comme TCP_CA_Open.

Disorder : Quand l'émetteur détecte un dupACK, il passe en état Disorder. Pendant cet état la fenêtre de congestion reste inchangée, mais chaque paquet arrivé déclenche l'envoi d'un nouveau segment. Néanmoins un paquet n'est pas envoyé tant que les paquets précédents n'ont pas quitté le réseau. En Linux cet état est défini comme TCP_CA_Disorder.

CWR : TCP peut recevoir des notifications ECN. Quand l'émetteur reçoit un avis de congestion, il ne réduit pas sa fenêtre de congestion en une seule fois mais par un segment à chaque arrivée d'un deuxième acquittement et ceci jusqu 'à ce que la taille de la cwnd soit diminuée de moitié. Si pendant ce processus de diminution de la fenêtre de congestion, il n'y a pas de retransmission de paquets déjà émis et non acquittés, alors l'émetteur est en état CWR (**Congestion Window Reduced**). L'état CWR peut être interrompu par un nouvel état Recovery ou Loss. En Linux cet état est défini comme TCP_CA_CWR.

Recovery : Après l'arrivée de plusieurs acquittements dupliqués successifs (définis par la constante `TCP_FASTRETRANS_THRESH` déclarée dans `/include/linux/tcp.h`, en général initialisée à 3, valeur recommandée par les spécifications du contrôle de la congestion) l'émetteur « retransmet » le premier segment attendu par le récepteur et se met en état Recovery.
Tant que TCP est en cet état, la réception de chaque deuxième dupACK provoque la diminution de la taille de la cwnd d'un segment. La réduction de la fenêtre s'arrête quand la taille de la fenêtre est égale au ssthresh c'est-à-dire la moitié de la fenêtre au moment du passage à l'état Recovery. La fenêtre de congestion n'augmente pas pendant cet état et le TCP envoie soit des segments réclamés par le récepteur soit des nouveaux paquets.
Le TCP reste en état Recovery jusqu'à ce que tous les segments en attente au moment du passage à l'état soient envoyés avec succès au récepteur c'est-à-dire qu'un acquittement cumulatif global (fullACK) soit reçu. Ensuite le TCP passe en état Open. Seul un timeout peut interrompre un Recovery. En Linux cet état est défini comme TCP_CA_Recovery.

Loss : Quand un RTO expire, le TCP passe en état Loss. Tous les segments en attente sont marqués comme perdus et la taille de la fenêtre de congestion passe à 1 segment. Désormais TCP commence à augmenter la taille de la fenêtre de congestion avec l'algorithme de slow-start. La différence majeure entre les états Recovery et Loss est qu'à l'état Loss, la fenêtre de congestion est

augmentée après que le TCP l'a initialisée à 1, alors qu'en Recovery la fenêtre peut seulement diminuer.
Aucun autre état ne peut interrompre le processus de l'état Loss. Ainsi TCP passe à l'état Open seulement lorsque tous les segments en suspens depuis le passage à l'état Loss n'ont été reçus par le récepteur. En Linux cet état est défini comme TCP_CA_Loss.

5.3.5 Retransmit et « burst data »

TCP évite les appels explicites pour transmettre un paquet dans n'importe lequel des états mentionnés ci-dessus, par exemple, concernant le Fast Retransmit. L'état du contrôle de congestion en cours détermine la manière dont la fenêtre de congestion est ajustée, et si l'émetteur doit considérer les segments non acquittés comme perdus.

Après que TCP ait traité un acquittement entrant selon l'état dans lequel il se trouve, il transmet des segments tant que `in_flight < cwnd`. L'émetteur retransmet le premier segment considéré comme perdu et pas encore retransmis, ou transmet des nouveaux segments s'il n'y a aucun paquet perdu en attente de retransmission.

Dans certaines circonstances le nombre de paquets « outstanding » diminue soudainement de plusieurs segments. Par exemple, un segment retransmis et les transmissions suivantes peuvent être acquittés avec un simple fullACK. Ces situations peuvent provoquer la transmission de rafales des données (**burst data**) sur le réseau, à moins qu'elles ne soient prises en considération dans l'implémentation de l'émetteur TCP. Ainsi, Linux évite les rafales en limitant la fenêtre de congestion pour permettre la transmission de tout au plus trois segments pour un ACK arrivé. La prévention des rafales peut réduire la taille de la fenêtre de congestion au-dessous du ssthresh. Il est alors possible que l'émetteur passe en slow-start après que plusieurs segments aient été acquittés par un ACK.

5.3.6 Le cache

Quand une connexion TCP est établie, les variables de TCP sont initialisées avec des valeurs fixes. Cependant, afin d'améliorer l'efficacité de communication au début de la connexion, Linux copie dans le cache de la destination, le maximum segment size utilisé, le ssthresh, les variables utilisées pour le calcul du RTO et un estimateur qui calcule la probabilité de réordonnancement après chaque connexion TCP. Si une autre connexion est établie avec la même adresse IP de destination que celle qui se trouve dans le cache, les valeurs stockées peuvent être utilisées pour initialiser la nouvelle connexion TCP. Mais si la qualité du réseau entre l'émetteur et le récepteur change pour une raison quelconque, les valeurs dans le cache

de la destination deviendraient momentanément périmées. Cependant, ceci est considéré comme un inconvénient mineur.

5.3.7 Différences par rapport aux spécifications TCP

Bien que Linux se conforme aux règles générales de contrôle de congestion de TCP définies par RFC 2582[6], il adopte parfois des techniques différentes pour effectuer certaines tâches du contrôle de la congestion [4].

Par exemple, pour décider du nombre de segments à transmettre, il serait logique de comparer la taille de la fenêtre de congestion à la différence de position du bord droit par rapport au bord gauche de la fenêtre glissante (variables `snd_una` et `snd_nxt` dans `tcp_sock`). Au lieu de cela, Linux évalue le nombre de paquets actuellement considérés comme « émis mais non acquittés » (voir section 5.3.2) dans le réseau. Ensuite il compare le nombre de ces segments à la taille de la fenêtre de congestion et prend des décisions sur le nombre de segments à transmettre.

De la même manière, avant le lancement des algorithmes slow-start et congestion avoidance, Linux compare le nombre des paquets `in_flight` (voir section 5.3.2) par rapport à la taille de la fenêtre de congestion. Si `in_flight < cwnd`, Linux termine l'algorithme, sinon la fenêtre de congestion évolue selon qu'elle se trouve en phase slow-start ou en congestion avoidance. Ceci peut paraître logique lorsqu'on constate que pendant tout le contrôle de la congestion Linux maintient la taille de la fenêtre de congestion en la comparant à la quantité des paquets `in_flight` et la taille de la cwnd. Si cette dernière est supérieure à `in_flight + 1`, alors la fenêtre de congestion est ramenée à la valeur de la variable `in_flight + 1`.

Autre exemple, Linux évalue le nombre de segments « outstanding » en unités de paquets de taille égale à MSS, tandis que les spécifications TCP évaluent la fenêtre de congestion en nombre d'octets transmis. Ceci a comme conséquence un comportement différent dans le cas des petits segments de données à envoyer : si l'implémentation emploie une fenêtre de congestion mesurée en octets (byte-based), elle permet à plusieurs petits paquets d'être envoyés sur le réseau, pour chaque segment de taille MSS dans la fenêtre de congestion. Linux, de son côté, permet la transmission d'un seul paquet pour chaque segment compté dans la fenêtre de congestion, indépendamment de sa taille. Par conséquent, Linux est plus conservateur comparé à l'approche byte-based, quand la charge utile de TCP se compose de petits segments.

Comme expliqué dans [4] section 4, il est ainsi très possible dans certaines situations qu'un 3dupACK ne déclenche pas un Fast Retransmit. Aussi, quand il entre en Fast Recovery, l'émetteur TCP de Linux ne réduit pas la fenêtre de congestion comme décris à la section 2.2.2. La fenêtre de congestion est diminuée d'un segment à la réception d'un acquittement

sur deux, jusqu'à ce que la taille de la fenêtre de congestion atteigne la moitié de sa valeur au moment du déclenchement de l'algorithme de Fast Recovery (Rate Halving).

Le calcul du RTO diffère aussi à celui de la spécification définie par l'IETF. Linux utilise une limite RTO de 200ms au lieu d' 1 sec définie par RFC2988.

5.4 Les dispositifs particuliers de Linux

Certains dispositifs choisis par Linux diffèrent d'une implémentation typique de TCP. Linux implémente un certain nombre de perfectionnements de TCP, tel que la ECN [25] et D-SACK [24]. Ces dispositifs ne sont pas encore largement déployés dans des réalisations de TCP, mais sont susceptibles de l'être à l'avenir parce qu'ils sont promus par l'IETF. En voici une liste non exhaustive [4] :

- Calcul du timer de retransmission,
- Annulation de la mise à jour de la fenêtre de congestion,
- Acquittements retardés,
- Validation de la fenêtre de congestion,
- Explicit Congestion Notification (ECN).

5.5 Le code source de Linux

Dans le code source du kernel 2.6.16-13 les programmes nécessaires à la gestion du réseau et plus particulièrement de la couche de transport se trouvent pour la plupart dans le répertoire `usr/src/linux-2.6.16-13/net/ipv4`. Certains fichiers comme les headers `net/tcp.h` et `linux/tcp.h`, très importants parce qu'ils contiennent la description des structures sock, liés à l'algorithme de gestion de la congestion et au Linux' Pluggable Congestion Control Algorithm se trouvent dans `usr/src/linux-2.6.16-13/include/`. La figure 18 présente l'arborescence des différents répertoires qui composent la structure du code du kernel. Les répertoires qui sont concernés par l'implémentation des modules de contrôle de la congestion y sont indiqués en gras.

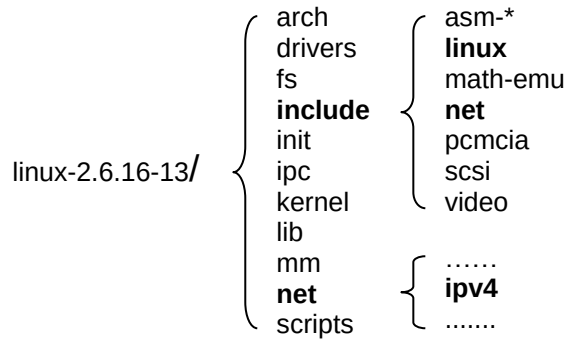


Figure 18: Networking code in the Linux kernel tree [3]

5.5.1 Description de l'implémentation de TCP

Cette partie de la gestion du réseau constitue la partie la plus complexe du code source dans le kernel de Linux.

TCP contribue à la grande majorité du trafic de données. Il accomplit deux fonctions importantes :

1) il établit une connexion fiable entre deux extrémités, un émetteur et un récepteur en retransmettant les paquets non acquittés et
2) il implémente le contrôle de la congestion en réduisant le débit envoyé quand une congestion est détectée.

Bien que les deux extrémités puissent être à la fois émetteur et récepteur, la présentation du code se fera selon le comportement de l'émetteur, quand il envoie des données, reçoit des acquittements du récepteur, retransmet les paquets perdus et adapte le débit d'émission en réduisant la fenêtre de congestion.

Les fichiers du code source qui contiennent l'ensemble des routines utilisées par TCP sont :

- tcp_input.c : code qui prend en charge des paquets en provenance du réseau.
- tcp_output.c : code qui s'occupe des paquets à envoyer sur le réseau.
- tcp.c : code TCP général. Il fait le lien avec la couche IP, via le socket, et contient des fonctions importantes pour établir et clôturer des connexions.
- tcp_ipv4.c : code spécifique pour le TCP IPv4 (existe aussi pour IPv6).
- tcp_timer.c : cette partie du code gère le timer.
- tcp.h : ce sont deux fichiers de définition des constantes TCP. Un dans `/include/linux` et l'autre dans `/include/net`.

Les détails du fonctionnement du code de la machine TCP de Linux, sont développés dans [3].

5.5.2 Les variables

L'algorithme de contrôle de congestion du Linux, Reno en l'occurrence, utilise un certain nombre de variables pour sa gestion, telles qu'elles sont définies par Van Jacobson dans RFC 793 [16]. Ces variables peuvent aussi être utilisées par les modules additionnels de contrôle de congestion. Les plus importantes de ces variables sont décrites dans le paragraphe consacré à la structure sock, tcp_sock.

5.6 Les structures des données

Le kernel utilise en général, deux structures de données pour la gestion du réseau. Une pour conserver l'état d'une connexion et une autre pour conserver les données et l'état de paquets entrants et sortants. Ces deux structures sont explicitées ci-après.

5.6.1 La structure sock

L'état de la connexion de TCP est stocké dans une structure appelée **sock** (pour socket).

sock est une entité indépendante du protocole dans laquelle se trouvent des informations concernant :

- la source et les adresses de destination utilisées pour la connexion,
- un pointeur vers le cache de la destination pour aider à trouver plus efficacement la destination d'un paquet, et
- d'autres données diverses au sujet de l'état de la connexion.

sock réserve également une sous-structure, dans laquelle sont stockées des informations relatives à l'état de la connexion TCP : la structure **tcp_sock**. Cette sous-structure contient des informations sur la taille de la fenêtre de congestion cwnd, la taille maximum de segment (maximum segment size), les round-trip-time (RTT), les compteurs de retransmission, et diverses statistiques. En fait, la plupart des variables référencées dans ce chapitre sont stockées dans cette partie de la structure de sock.

5.6.2 La structure sk_buff

Linux utilise des socket buffers pour passer des données entre les couches des protocoles et les drivers des cartes réseau. Le but de cette « bufferisation » est la connaissance par la couche en cours de ce qui a été ajouté dans l'en-tête du paquet par la couche précédente.
Cette structure, appelée **sk_buff,** est définie dans `include/linux/skbuff.h`. Chaque paquet appartenant à un socket est contenu dans cette structure. Ainsi, une structure de données sk_buff est créée chaque fois qu'un paquet est traité par le kernel, qu'il provienne de l'application ou du réseau,. Toute modification des données du paquet implique une modification dans un des champs de cette structure. Souvent, cette structure est considérée comme paramètre pour des fonctions de réseau (sous le nom **skb**).

Chaque socket a des files d'attente destinées aux sk_buff entrants et aux sk_buff sortants. Les deux premiers champs de sk_buff sont des pointeurs vers le sk_buff précédent et vers le sk_buff suivant. **sk_buff_head** pointe vers l'en-tête de la liste des sk_buff. Ainsi, les sk_buff sont liés entre eux par des pointeurs, telle une liste circulaire bidirectionnelle, permettant un accès facile aux paquets.

En plus de l'en-tête et des données du paquet, le sk_buff contient une table de contrôle, qui inclut divers indicateurs et identifiants du protocole. Le socket n'a pas de buffer spécifique pour des données, entrantes ou sortantes, mais les tailles des sk_buff sont enregistrées en utilisant une des options du socket qui définit une limite maximum du nombre de sk_buff qui peuvent être liés au socket. En adoptant cette approche, le nombre d'opérations de copie consommatrices en mémoire peut être réduit au minimum. La figure 19, montre bien la relation de structures sock et sk_buff.

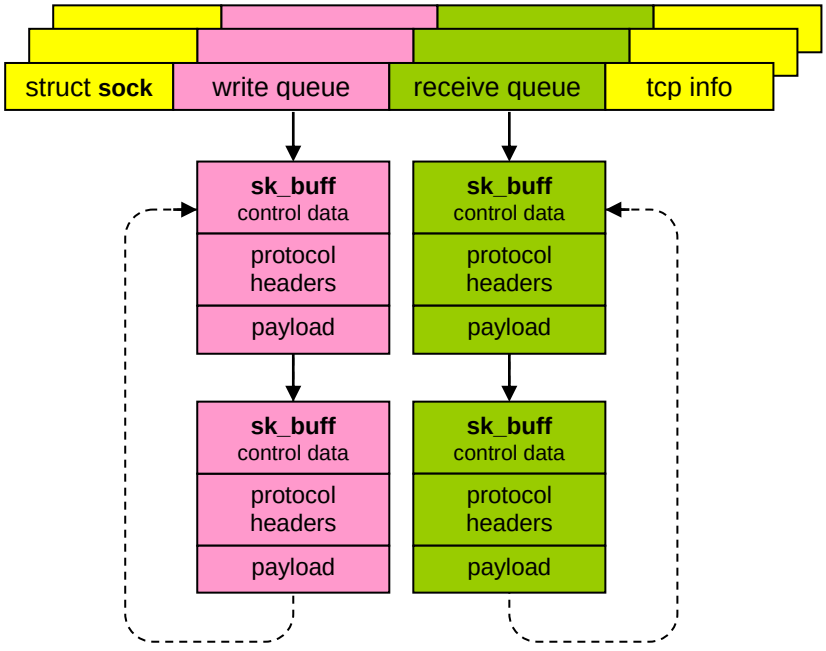


Figure 19 : socket et sk_buff [4]

Il y a trois files d'attente différentes pour stocker des sk_buffs :

- **write_queue** stocke les données écrites par l'application non encore acquittées par le récepteur.
- **receive_queue** stocke les paquets reçus du réseau et non encore lus par l'application.
- **out-of-order_queue** (non représentée à la figure 18) stocke les paquets en provenance du réseau, qui ne peuvent pas être livrés à l'application parce qu'une partie des données est absente.

Ainsi, quand une application envoie des données, un nouveau sk_buff est créé et les données « utiles » remplissent le secteur qui leur est réservé (le **payload**). Un espace est réservé pour les en-têtes de TCP, de l'IP et des couches inférieures. Les indicateurs nécessaires sont mis à jour à un endroit prévu, au début du sk_buff.

Les en-têtes du protocole sont remplis selon les données dans la structure sock, avant que le paquet ne soit passé aux couches inférieures pour être transmis. Si le payload n'a pas encore atteint sa taille maximum, il peut être complété par de nouvelles données reçues de l'application, aussi longtemps que l'émetteur TCP n'a pas transmis le segment.

5.6.3 La structure tcp_sock

tcp_sock est l'une des principales structures de sock. Des informations concernant l'état du contrôle de la congestion y sont enregistrées. Parmi toutes les variables de la structure, certaines sont importantes pour la réalisation du module TCP Xeno :

- snd_nxt : numéro de séquence du premier byte du segment à envoyer,
- snd_una : numéro de séquence du premier byte du segment dont TCP attend l'acquittement,
- snd_ssthresh : seuil du slow-start,
- snd_cwnd : la fenêtre de congestion,
- snd_cwnd_cnt : fraction de la fenêtre de congestion, utilisé à la phase AI (tcp_cong_avoid),
- snd_cwnd_clamp : limite supérieure de la fenêtre de congestion,
- snd_cwnd_stamp : instant où la fenêtre de congestion a été modifiée pour la dernière fois,
- bytes_acked : nombre de bytes acquittés par le dernier ACK,
- high_seq : maximum sequence number,

ainsi que les compteurs décrits à la section 5.3.1 :

- packets_out : nombre des paquets envoyés sur le réseau,
- left_out : nombre de paquets qui ont quitté le réseau,
- retrans_out : nombre de paquets retransmis,

- sacked_out : nombre des paquets SACKed, et
- fackets_out : nombre des paquets FACKed.

La description complète de la structure peut être consultée sur [21].

5.7 Le Linux' Pluggable Congestion Control Algorithm

Implémenter un nouvel algorithme de contrôle de congestion TCP dans le kernel de Linux, nécessite une connaissance approfondie du fonctionnement du système et des spécifications TCP qui y sont intégrées. Dans cette rubrique sera décrite une nouvelle méthode d'intégration de modules de contrôle de la congestion qui vient d'être réalisé depuis la version 2.6.16.
En utilisant cette méthode, il est possible d'implémenter un module de contrôle de congestion d'une manière assez simple et directe, en évitant de devoir modifier le code du kernel de Linux.

5.7.1 Introduction

Pour aider ceux qui expérimentent des nouvelles méthodes d'évitement de la congestion et afin de ne pas devoir programmer directement dans le code du kernel, *Stephen Hemminger* a conçu une nouvelle structure de codage qui permet à des algorithmes d'évitement de la congestion d'être introduits en tant que modules que l'on peut charger dans Linux au moment voulu.

Pour implémenter cette nouvelle structure de code il a également retouché les algorithmes existants du kernel. Tout son travail a été édité après la version 2.6.16 du kernel de Linux.

Une interface commune pour les algorithmes de contrôle de congestion a été définie. Ainsi les développeurs peuvent implémenter facilement leurs algorithmes comme modules du Linux [14] [15].

5.7.2 L'interface de l'algorithme de contrôle de la congestion

Un nouveau mécanisme de contrôle de congestion peut être enregistré à l'aide des fonctions dans `tcp_cong.c`. Les fonctions utilisées par ce mécanisme de contrôle de congestion sont enregistrées par `tcp_register_congestion_control` par l'intermédiaire d'une structure définie dans `/include/net/tcp.h`, la `tcp_congestion_ops` dont la description est la suivante :

```
#define TCP_CA_NAME_MAX       16
struct tcp_congestion_ops {
        struct list_head          list;
                        /* initialize private data (optional) */
        void (*init)(struct sock *sk);
                        /* cleanup private data  (optional) */
        void (*release)(struct sock *sk);
                        /* return slow start threshold (required) */
        u32 (*ssthresh)(struct sock *sk);
                        /* lower bound for congestion window (optional) */
        u32 (*min_cwnd)(struct sock *sk);
                        /* do new cwnd calculation (required) */
        void (*cong_avoid)(struct sock *sk, u32 ack, u32 rtt, u32 in_flight, int good_ack);
                        /* round trip time sample per acked packet (optional) */
        void (*rtt_sample)(struct sock *sk, u32 usrtt);
                        /* call before changing ca_state (optional) */
        void (*set_state)(struct sock *sk, u8 new_state);
                        /* call when cwnd event occurs (optional) */
        void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);
                        /* new value of cwnd after loss (optional) */
        u32  (*undo_cwnd)(struct sock *sk);
                        /* hook for packet ack accounting (optional) */
        void (*pkts_acked)(struct sock *sk, u32 num_acked);
                        /* get info for inet_diag (optional) */
        void (*get_info)(struct sock *sk, u32 ext, struct sk_buff *skb);

        char   name[TCP_CA_NAME_MAX];
        struct module *owner;
};
```

Chacune des fonctions comprises dans cette structure est un « crochet » dans le code de TCP qui permet à l'algorithme d'obtenir des informations sur l'état du réseau et de réagir en conséquence :

init() : cette fonction est appelée à l'arrivée du premier ACK reçu d'une nouvelle série de transmissions, et avant que le mécanisme AIMD ne soit appelé pour la première fois. Si l'algorithme utilise la zone `icsk_ca_priv`, elle doit être initialisée ici.

release() : si l'algorithme de contrôle de la congestion a réservé plus de mémoire additionnelle que les 16 unsigned int de `icsk_ca_priv`, il doit libérer la mémoire additionnelle occupée pour éviter des débordements de mémoire.

ssthresh() : cette fonction est appelée quand une perte est détectée. Elle renvoie une nouvelle valeur du ssthresh, calculé par l'algorithme.

min_cwnd() : cette fonction est souvent appelée après le `ssthresh().min_cwnd()` fournit, en cas de réduction de la fenêtre de congestion, la valeur minimum en dessous de laquelle l'algorithme ne doit pas descendre. Souvent elle retourne la valeur du ssthresh courant. Mais cette fonction peut être utilisée aussi pour forcer un slow-start après un recovery si c'est nécessaire pour l'algorithme.

cong_avoid() : cette fonction est appelée quand un acquittement arrive de l'autre extrémité et que la fenêtre de congestion peut être incrémentée selon des règles définies par l'état de l'algorithme, si certaines conditions définies par l'algorithme sont respectées.

rtt_sample() : cette fonction est appelée quand une nouvelle mesure d'un temps aller-retour (RTT) est arrivée. La valeur en retour est exprimée en microsecondes.

set_state() indique que l'état de congestion de TCP a changé. La variable concernée, `tcp_ca_state`, se trouve dans `/include/linux/tcp.h`.

cwnd_event() indique à l'algorithme que divers événements intéressants se sont produits. Les événements qui peuvent se produire sont définis dans `/include/net/tcp.h` (tcp_ca_events) :

- CA_EVENT_FAST_ACK : un ACK en séquence est arrivé,
- CA_EVENT_SLOW_ACK : un ACK hors séquence est arrivé,
- CA_EVENT_TX_START : Première transmission quand in_flight = 0 ;
- CA_EVENT_CWND_RESTART : La fenêtre de congestion est relancée,
- CA_EVENT_COMPLETE_CWR : Le recovery de la fenêtre de congestion a terminé,
- CA_EVENT_FRTO : un Timeout pendant le Fast Recovery est arrivé,
- CA_EVENT_LOSS : un Timeout pendant une retransmission est arrivé.

Parfois, des situations passagères nécessitent la réduction de la fenêtre de congestion. La méthode **undo_cwnd**() est appelée quand une telle procédure est annulée, souvent pour reconstituer une fenêtre de plus grande taille.

pkts_acked() : cette fonction est appelée dans le cas d'un ACK qui acquitte certains nouveaux paquets.

get_info() est utilisé pour fournir à la user space certaines informations disponibles sur l'état de l'algorithme d'évitement de congestion.

Quatre fonctions de la structure doivent être obligatoirement implémentées dans le module pour assurer les fonctionnalités de base pour le contrôle de congestion : `init()`, `ssthresh()`, `min_cwnd()` et `cong_avoid()`.

Un espace a été prévu pour le stockage par le mécanisme de ces variables propres. Il s'agit d'un vecteur de 16 x u32, 512 bits, implémenté comme `icsk_ca_priv` [21].

Aussi, `icks_ca_ops`, est un pointeur vers l'interface de l'algorithme de contrôle de la congestion. `icsk_state` indique l'état courant du contrôle de congestion : TCP_CA_Open, TCP_CA_Disorder, TCP_CA_CWR, TCP_CA_Recovery et TCP_CA_Loss.

Pour implémenter un algorithme de contrôle de congestion il faut suivre le processus suivant :

1) Comprendre la structure des données et la nature des fonctions de l'interface de contrôle de la congestion,
2) Donner un nom à son algorithme, p.ex. « xeno »,
3) Créer une structure de variables propre à l'algorithme,
4) Implémenter les fonctions nécessaires à l'interface, au moins les quatre obligatoires,
5) Donner un nom à son programme dans /net/ipv4, p.ex. tcp_xeno.c
6) Mettre à jour le fichier Makefile en respectant la structure requise du fichier.
7) Compiler, et ensuite exécuter et tester son module.

5.8 Le module tcp_xeno.c

Ce paragraphe est une brève explication de la structure du module. Le code source de **tcp_xeno.c** se trouve dans l'annexe de ce document.

L'implémentation de ce module nécessite l'insertion dans le kernel de quelques lignes de code, en raison de contraintes imposées par Linux. Suite à cela, la modularité décrite c-dessus n'a pas pu être pleinement mise en œuvre.

Les décompteurs **`forced_trans_ctr`** et **`ignore_data_ctr`**, ainsi que le switch **`is_dupack`**, dont l'utilité sera expliquée plus loin, y sont définis et initialisés. Le code source de ces ajouts se trouve aussi dans l'annexe avec le code source du module.

5.8.1 Le paramètre grate

Le paramètre **grate** est une valeur qui reste fixe pendant tout le processus. grate représente $rate_G$, le débit à garantir par l'algorithme. Dans [2] le $rate_G$ est exprimé en paquets/sec. Cette unité est propre à TCP, les applications expriment souvent le débit en kbits/sec (kbps).

Comme déjà présenté au paragraphe 5.3.7, une des différences entre la gestion de la congestion de Linux et le TCP en général, est le mode de calcul de la taille de la fenêtre de congestion. En Linux la taille de la fenêtre de congestion est estimée en segments. Pour ne pas compliquer le calcul, le paramètre grate représente des segments/sec.

Néanmoins, l'adaptation du code pour la conversion d'un grate fourni en Kbps, en segments/sec est possible. Le MSS est stocké dans la variable mss_cache de la structure tcp_sock.

5.8.2 Le calcul de BaseRTT

BaseRTT est mis à jour à chaque fois qu'un RTTsample, fourni par la fonction `tcp_xeno_rtt_sample()`, est plus petit que le BaseRTT existant.

5.8.3 Le calcul de $cwnd_G$

- La fenêtre de congestion cwnd est égale à la somme de ces deux composantes cwndG et cwndBE. Alors,

$$cwnd = (rate_G * BaseRTT) + cwnd_{BE}$$

- BaseRTT est fourni par Linux en microsecondes (µsec = 10^{-6} sec), alors le calcul de la partie service garanti de la fenêtre de congestion devient,

$$cwnd_G\ [segments] = (rate_G\ [segments/sec] * BaseRTT\ [\mu sec]\ )\ /\ 10^6$$

- $cwnd_G$ est mis à jour chaque fois que BaseRTT est mis à jour. Cela signifie que $cwnd_G$ n'est pas une valeur fixe, il varie en fonction de BaseRTT pendant tout le processus. Ainsi les figures 13 et 14 représentent $cwnd_G$ pour un BaseRTT donné.

5.8.4 Le calcul du ssthresh

- $cwnd_{BE} = cwnd - cwnd_G$
- Si $N < \beta$ la fonction tcp_xeno_sstresh() returne ⅘ $cwnd_{BE}$ sinon elle retourne ½ $cwnd_{BE}$.

5.8.5 Les états de l'algorithme de tcp_xeno

L'algorithme de tcp_xeno, selon l'état du contrôle de la congestion de TCP, peut se trouver dans 3 états différents :

- NO_LOSS_DEFAULT_TCP_CA_BEHAVIOUR : tcp_xeno passe dans cet état quand le TCP passe en état TCP_CA_Open. Les decompteurs `forced_trans_ctr`, `ignore_data_ctr` et le switch `is_dupack` sont initialisées à 0. BaseRTT peut être recalculé.

- XENO_FASTREC_HANDLING_MECHANISM : tcp_xeno passe dans cet état quand le TCP passe en état TCP_CA_Recovery. Il initialise le décompteur, `forced_trans_ctr`, à $cwnd_G$. L'autre decompteur, `ignore_data_ctr`, est initialisé à ½ $cwnd_{BE}$ ou à ⅘ $cwnd_{BE}$ selon que le réseau est estimé en congestion ou pas. Pas de calcul de BaseRTT durant cet état.

- XENO_TIMEOUT_HANDLING_MECHANISM : tcp_xeno passe dans cet état quand le TCP passe en état TCP_CA_Loss. Il initialise le décompteur `forced_trans_ctr` à $cwnd_G$. `ignore_data_ctr` est initialisé à `in_flight` − $cwnd_G$, si `in_flight` > $cwnd_G$. Pas de calcul de BaseRTT durant cet état.

5.8.6 Le rôle de min_cwnd()

Pour éviter que Linux en réduisant la cwnd à la réception de chaque dupACK, provoque un demarrage en slow-start, la fonction `tcp_xeno_min_cwnd()` retourne cwndG + sshtresh.

5.8.7 Les events

- CA_EVENT_TX_START : tcp_xeno est réinitialisé.
- CA_EVENT_CWND_RESTART : les variables de calcul de BaseRTT sont réinitialisées.
- CA_EVENT_COMPLETE_CWR : cwnd = $cwnd_{update}$. Linux initialise cwnd après le changement d'état en TCP_CA_Recovery. Alors il a été utile de mettre à jour cwnd après cet event.

5.8.8 Le congestion avoidance

Le calcul de la différence N est fait de la même manière que dans `tcp_veno.c`.

5.9 Comment charger le module sous Linux

L'algorithme de TCP Reno qui a été conçu par Van Jacobson reste toujours intégré dans le kernel de Linux mais il peut être « contourné » (overridden) par le module. Ensuite, le dernier module enregistré devient le mécanisme de contrôle de congestion par défaut (LIFO).

Le fichier `sysctl.conf` contient la liste des paramètres utilisés pour l'ensemble du système.

La commande `sysctl -a net.ipv4.tcp_congestion_control` informe du mécanisme utilisé pour le contrôle de congestion.

La commande `sysctl -w net.ipv4.tcp_congestion_control=xeno` permet de modifier le paramètre enregistré dans `sysctl.conf`.

Au démarrage du système, le fichier `sysctl.conf` est consulté et le mécanisme de contrôle de congestion associé à la variable `net.ipv4.tcp_congestion_control` est chargé en mémoire.

En cas de retrait du module de contrôle de congestion par défaut, le module suivant disponible sera chargé. Reno n'est pas un module, il est intégré dans le kernel et ne peut être supprimé. Ainsi, il est toujours disponible en dernier secours.

En utilisant cette nouvelle API de contrôle de congestion, l'implémentation de nouveaux algorithmes s'avère beaucoup plus simple de plusieurs points de vue :

- Primo, il n'est pas nécessaire de parcourir directement le code source du kernel. Chaque algorithme est empaqueté dans un module différent. Ainsi, l'algorithme de TCP Xeno est placé dans `tcp_xeno.c`.

- Secundo, remplacer le module en cours devient très simple. Une fois que les modules sont compilés dans le kernel et installés, il suffit d'utiliser la commande `modprobe` pour charger le module souhaité. Ainsi, en faisant `modprobe tcp_xeno`, TCP Xeno devient le module de contrôle de la congestion courant pour Linux.

- Et tertio, si l'API ne change pas, les modules déjà développés sont indépendants du développement du corps de kernel. Ainsi ils pourront être intégrés sans aucune modification dans les versions futures du kernel.

5.10 Limitations de l'implémentation du module TCP Xeno.

Comme expliqué précédemment, Linux offre des possibilités de développement de modules destinés à implémenter de nouveaux algorithmes de contrôle de la congestion. Toute cette infrastructure est en grande partie basée sur les spécifications (RFC) publiées depuis des années par l'IETF. C'est ainsi qu'il a été décidé d'implémenter TCP Xeno comme module additionnel au contrôle de congestion de Linux, pour confirmer l'approche théorique du chapitre 4 par des tests en situation réelle.

Néanmoins, le fait que le protocole de départ GTCP ne respecte pas l'ordre d'exécution de l'algorithme spécifié dans le RFC3782 [7] relatif au fonctionnement de l'algorithme de contrôle de congestion, notamment au niveau de la gestion de la fenêtre de congestion au moment du Fast Recovery, nous oblige à devoir implémenter le protocole dans des systèmes adaptés au mécanisme de refus de transmission de paquets pendant la phase Fast Recovery – ignore data. Ce qui rend le produit de ce travail non-modulable pour Linux.

De plus, Linux, pendant la phase Fast Recovery, gère de manière optimale le « gonflement – dégonflement » de la fenêtre de congestion, défini dans le RFC 3782, sans passer par l'algorithme de congestion avoidance à la réception d'un acquittement dupliqué pour « gonfler » la fenêtre de congestion, probablement par respect des spécifications RFC 3042 [28], éliminant ainsi la possibilité de comptage des dupACK arrivés, à l'intérieur du mécanisme d'évitement de congestion. Cette fonctionnalité a été implémentée à l'intérieur du kernel lui-même.

Tout cela étant dit, les modifications apportées au code du kernel, pour assurer sa compatibilité avec TCP Xeno sont mineures. Ces modifications ne perturbent en rien le bon fonctionnement du code dans son comportement habituel et n'empêchent pas l'utilisation du mécanisme de base Reno, ni des autres modules implémentés. Ce Linux est, tout simplement, devenu « TCP Xeno compatible ». De plus, les paramètres implémentés peuvent être utilisés par d'autres modules de la même conception que Xeno.

Chapitre 6

Evaluation

6.1 Description de l'équipement des tests

Les tests ont été effectués sur un seul ordinateur de type Pentium 4. Le logiciel VMware 5.x y a été installé. Un deuxième ordinateur avec Linux Ubuntu a été utilisé comme serveur FTP.

Seul le code source de TCP Veno est disponible sur Internet, prévu pour la version 2.6.16.x de Linux. C'est la raison pour laquelle Suse 10.1 a été choisi. En effet, son kernel est de version 2.6.16-13.

Par contre, malgré de multiples tentatives, il ne nous a pas été possible de nous procurer une implémentation de GTCP, ou de recevoir celle-ci des auteurs. GTCP a donc été complètement réimplémenté sur base de sa description fonctionnelle dans [2].

Suse 10.1 a été installé dans VMware 5. Le simulateur netem [31] et l'analyseur Ethereal [32] font partie de l'ensemble des logiciels que Suse 10.1 installe en plus du système de base.

6.2 Tests d'évaluation des performances et résultats

Dans cette section, deux points sont abordés : la validation et la démonstration des performances de TCP Xeno.

Pour la validation, il importe de montrer que l'implémentation Linux de TCP Xeno se comporte comme attendu dans les cinq scénarios de perte définis à la section 4.1, pour autant qu'il soit possible de générer ces cinq scénarios.

Pour l'étude de performance, il s'agit de comparer la vitesse de transfert de TCP Xeno à celle de TCP Reno et de TCP Veno.

6.2.1 La mise en place des scénarios.

Pour pouvoir tester les 5 scénarios possibles de perte, on doit pouvoir :

- provoquer des 3dupACK,

- créer les conditions pour lesquelles $N < \beta$ ou $N \geq \beta$,
- provoquer un nombre défini (k) de pertes pendant un Fast Recovery,
- provoquer des timeout.

Les deux extrémités de la connexion étant assez proches, les RTT mesurés seront très courts. Comme le calcul du RTO est basé sur ces RTT il ne sera pas évident de provoquer des 3dupACK avant l'expiration du RTO. Cependant en simulant des longs délais de communication on peut allonger les RTO. On peut définir de tels délais avec le simulateur netem. Ces délais sont exprimés en millisecondes.

Le simulateur netem offre aussi la possibilité de générer un pourcentage défini de pertes. On peut espérer provoquer des dupACK et des situations de pertes légères ($N < \beta$) en introduisant un pourcentage de ±10% de pertes dans netem.

En introduisant des pourcentages élevés de pertes à netem, on peut provoquer des timeout. Mais attention, un pourcentage élevé de pertes peut provoquer le blocage de la communication.

Avec netem il n'y a pas moyen de provoquer un nombre défini de pertes (k) pendant un Fast Recovery, s'il y en a.

La taille de la largeur de bande à garantir est introduite par le paramètre fourni à TCP Xeno manuellement, émulant ainsi le mécanisme de service garanti du QoS.

6.2.2 Le test de validité

Le but de ce test est de vérifier les variations de la taille de la fenêtre de congestion et du ssthresh en fonction du scénario présenté. Les comportements attendus sont présentés dans la table ci-dessous :

scénario	cwnd	ssthresh
1	$cwnd_G + \frac{4}{5} cwnd_{BE}$	$\frac{4}{5} cwnd_{BE}$
2	$cwnd_G + \frac{1}{2} cwnd_{BE}$	$\frac{1}{2} cwnd_{BE}$
3	$cwnd_G$	$\frac{1}{2} cwnd_{BE}$
4	1	$\frac{1}{2} cwnd_{BE}$
5	1	$\frac{4}{5} cwnd_{BE}$

Le test consiste à transférer un fichier de 55 MB vers le serveur FTP. Un délai de 100ms a été introduit par netem ainsi qu'un taux de 10% de pertes. Des messages introduits à des endroits choisis du code ont permis de suivre le parcours de l'algorithme durant le transfert.

Résultat : Seuls les scénarios 4 et 5 ont pu être vérifiés. Le scénario 4 a pu être produit en débranchant un court instant la connexion Ethernet.

Ceci peut être expliqué par :

1) Le comportement particulier de Linux dans certains cas, décrit à la section 5.3.7.
2) L'absence de production d'un débit permanent à garantir, conforme au paramètre grate, pendant tout le transfert (à moins de définir un grate qui génère une $cwnd_G = 1$, ce qui n'apporte rien comme résultat de validation).
3) L'absence de 3dupACK. Malgré le taux de pertes, Linux n'a généré aucun 3dupACK et n'a déclenché aucun Fast Recovery.

Ce résultat ne signifie absolument pas que l'algorithme n'apporte pas les améliorations définies dans le chapitre 4. La démonstration de celles-ci nécessite cependant la mise au point d'un banc de tests permettant de créer les conditions propices à la validation complète de l'algorithme.

6.2.3 Le test de performance

La caractéristique principale à mesurer dans ce test est le débit. Le test consiste à transférer le fichier de 55 MB vers le serveur FTP et mesurer le temps et le débit moyen du transfert. Pour que TCP Xeno soit dans la situation pour laquelle il a été conçu, un délai de 100ms et un taux de perte de 10% ont été introduits pour les tests de 3 protocoles. Les résultats sont ceux affichés par FTP à la fin du transfert :

Test 1 : protocole TCP Reno
Résultat : 56913972 bytes sent in 33 :21 (27.76 KB/s)

Test 2 : protocole TCP Veno
Résultat : 56913972 bytes sent in 32 :51 (28.19 KB/s)

Test 3 : protocole TCP Xeno avec largeur de bande garantie de 33 paquets/sec.
Résultat : 56913972 bytes sent in 29 :01 (31.92 KB/s)

On constate que TCP Xeno offre un débit moyen de 12% supérieur à TCP Veno et 15% supérieur à TCP Reno.

Chapitre 7

Conclusions

7.1 Conclusion

Dans ce mémoire a été évoquée la problématique de l'amélioration des performances de protocoles de gestion de la congestion dans des réseaux avec des services garantis de largeur de bande. Une version améliorée du protocole de transport TCP a été proposée, appelée TCP Xeno. Dans les conditions envisagées, elle réalise des performances globales optimales sans compter les overheads additionnels par rapport à TCP. TCP Xeno implique des changements aux seuls mécanismes de commande de congestion de TCP, et par conséquent n'exige aucun changement au TCP récepteur.

Les tests de performance ont montré que dans un transfert avec pertes TCP Xeno offre un meilleur débit que TCP Reno ou TCP Veno.

7.2 Quelques remarques de l'auteur

La première partie du travail a nécessité la lecture de la documentation disponible sur TCP et ses variantes, suivie par l'étude théorique de la possibilité de combiner les deux mécanismes concernés, à savoir GTCP et TCP Veno.

La seconde partie, et de loin la plus importante, a consisté à mettre au point le module tcp_xeno. Pour la réalisation de ce travail il a été nécessaire, dans un premier temps d'acquérir un certain nombre de compétences techniques : utilisation du système d'exploitation de type Linux, apprentissage de la structure du kernel de ce système, au moins pour ce qui concerne la couche transport c'est-à-dire le TCP, et approfondissement des connaissances du langage C. Des outils appropriés à l'analyse du trafic réseau (Ethereal) et à la simulation des problèmes des grands réseaux (netem) ont été utilisés afin de pouvoir tester la robustesse de l'algorithme qui finit par apparaître à la fin de ce document.

N'ayant pas de connaissances approfondies en Linux, un grand nombre de documents a été consulté pour comprendre la structure du code source du kernel Linux, au niveau de TCP.

Avec comme seule aide le code source de Linux 2.6.16.13-4 fourni par Suse 10.1, le code de TCP Veno, publié entre-temps, et les documents [1] et [2], le module basé sur [2] a été constitué pour ensuite y introduire les spécifications Veno.

Une constatation à la fin de cette expérience est que le code source de Linux est loin d'être clair pour un novice, malgré la disponibilité des spécifications RFC et de quelques publications.

Où trouver un cahier de charges, avec description des différentes routines et de leurs variables ? Cela devrait être publié un jour. DataTAG [6] a tenté cette expérience en 2004, pour la partie TCP du kernel 2.4. Cela semble toutefois ne pas constituer un obstacle pour certains. Linux évolue sans cesse, grâce à la contribution de nombreux développeurs.

Bibliographie

[1] C. P. Fu, S. C. Liew, "***TCP Veno: TCP Enhancement for Transmission over Wireless Access Networks***," *IEEE (JSAC) Journal of Selected Areas in Communications,* Feb 2003. http://www.ntu.edu.sg/home/ascpfu/veno.pdf

[2] Y. Zhu, O. Oladeji and R. Sivakumar, "***Enhancing TCP for Networks with Guaranteed Bandwidth Services***," IEEE Global Communications Conference (GLOBECOM), San Francisco, CA, USA, December 2003. http://www.ece.gatech.edu/research/GNAN/archive/2003/globecom03zp.pdf, 20/08/2007

[3] Rio Miguel, "***A Map of the Networking Code in Linux Kernel 2.4.20***", Technical Report DataTAG-2004-1, 31 March 2004, www.datatag.org, 20/08/2007

[4] Pasi Sarolahti, Alexey Kuznetsov, "***Congestion Control in Linux TCP***", University of Helsinki, http://www.cs.helsinki.fi/research/iwtcp/papers/linuxtcp.pdf, 20/08/2007

[5] RFC 2581, http://www.codes-sources.com/rfc.aspx?rfc=2581, 20/08/2007

[6] RFC 2582, http://www.codes-sources.com/rfc.aspx?rfc=2582, 20/08/2007

[7] RFC 3782, http://www.codes-sources.com/rfc.aspx?rfc=3782, 20/08/2007

[8] TCP Vegas, http://www.cs.arizona.edu/projects/protocols/, 20/08/2007
http://www.opalsoft.net/qos/TCP-40.htm, 20/08/2007

[9] W. Richard Stevens, "***TCP/IP illustré – Les protocoles (Volume 1)*** ", Thomson publishing, 1996

[10] Andrew Tanenbaum, "***Réseaux***", 3e édition, DUNOD, 1996

[11] David X. Wei, Pei Cao, "***An NS-2 TCP Implementation with Congestion Control Algorithms from Linux***", http://www.cs.caltech.edu/%7Eweixl/technical/ns2linux/paper/wns2-final.pdf, 20/08/2007

[12] René Pfeiffer – "***TCP and Linux' Pluggable Congestion Control Algorithms***", http://linuxgazette.net/135/pfeiffer.html, 20/08/2007

[13] Xavier Dubois, "***Performance of Different TCP Versions over UMTS Common/Dedicated Channels***", FUNDP, Namur, 2005

[14] TCP Protocol, http://www-didc.lbl.gov/TCP-tuning/linux-2.6.13-tcp.txt, 20/08/2007

[15] http://lwn.net/Articles/128681/, 20/08/2007

[16] RFC 793, http://www.frameip.com/rfc/rfc793-fr.php, 20/08/2007

[17] Omar Ait-Hellal, Eitan Altman, "***Evolution of TCP: Problems and enhancements***", ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-3603.pdf, 20/08/2007

[18] Van Jacobson, "***Congestion Avoidance and control***", SIGCOMM 88, ACM, Aug 1988

[19] TCP Westwood Home Page, http://www.cs.ucla.edu/NRL/hpi/tcpw/, 20/08/2007

[20] Veno manual.pdf, http://www.ntu.edu.sg/home/ascpfu/veno/veno.html, 20/08/2007

[21] Le code source de Linux peut être obtenu dans http://www.kernel.org, 20/08/2007

[22] IETF, **Internet Engineering Task Force**, www.ietf.org, http://fr.wikipedia.org/wiki/Internet_Engineering_Task_Force, 20/08/2007

[23] RFC 2018, "***TCP Selective Acknowledgement Options***", http://www.faqs.org/rfcs/rfc2018.html, 19/08/2007

[24] RFC 2883, "***An Extension to the Selective Acknowledgement (SACK) option for TCP***", http://www.faqs.org/rfcs/rfc2883.html, 19/08/2007

[25] RFC 3168, "***Explicit Congestion Notification***", http://www.faqs.org/rfcs/rfc3168.html, 19/08/2007

[26] http://netlab.caltech.edu/FAST/references/Mo_comparisonwithTCPReno.pdf, 18/08/2007

[27] Talal Achkar Diab, Philippe Martins, "***Comparison of Eifel and standard versions of TCP over GPRS in the presence of radio link Disconnections***", http://www.enst.fr/_data/files/docs/id_485_1116921063_271.pdf, 17/08/2007

[28] RFC 3042, http://www.codes-sources.com/rfc.aspx?rfc=3042&p=2, 21/08/2007

[29] Constantinos Makassikis, "***Modèle de coût des communications TCP à un niveau applicatif***", Juin 2006, http://icps.u-strasbg.fr/upload/icps-2006-180.pdf, 28/08/2007

[30] Net100's TCP Vegas, http://www.csm.ornl.gov/~dunigan/net100/vegas.html, 31/08/2007

[31] netem http://developer.osdl.org/shemminger/netem/example.html, et http://linux-net.osdl.org/index.php/Netem, 02/08/2007

[32] Ethereal (Wireshark), http://lab.erasme.org/ethereal/, 02/08/2007