TCP Cours

## **1. Concepts de Base**

Initialisation connexion TCP : SYN - SYN ACK - ACK

* **Segment TCP** Un segment est l’unité de données (payload) transmise par TCP. Il ne s’agit pas d’un message complet, d’un paquet ou d'une frame. La taille maximale d’un segment (MSS – Maximum Segment Size) est négociée entre les hôtes. 1452 octets ou 1460
* **Fenêtres et Contrôle de Flux**
  + **Receiver Window (rwnd)** : Indique le nombre d’octets ou de segments que le récepteur peut accepter, utilisé pour le contrôle de flux.
  + **Congestion Window (cwnd)** : Représente le nombre de segments que le réseau peut supporter, utilisé pour le contrôle de congestion.
  + **Flight Size** : Le nombre de segments envoyés mais non encore acquittés. Notez que, contrairement à une idée répandue, Flight Size n’est égal à cwnd que lorsque tous les segments de la fenêtre ont été transmis.
* **Transmission et Séquence** Chaque segment TCP comporte un numéro de séquence. L’ISN (Initial Sequence Number) est envoyé lors du segment SYN et est incrémenté d’un pour chaque nouveau segment transmis. L’acquittement (ACK) correspond toujours au plus grand numéro de séquence reçu de manière continue.

Le **Round Trip Time (RTT)** en TCP correspond au temps entre l'envoi d'un segment de données et la réception de l'ACK correspondant.

* Il est utilisé pour déterminer si un segment est perdu ou non.
* Il peut varier considérablement en fonction des conditions du réseau.
* TCP calcule un estimateur de RTT qu'il met à jour à chaque réception d'un ACK.
* Une sous-estimation entraîne des retransmissions inutiles.
* Une surestimation provoque des temps d'attente trop longs avant de retransmettre un segment potentiellement perdu.

Le **Head-of-Line Blocking** est un problème qui se produit lorsqu'un paquet bloqué empêche la livraison des paquets suivants, même s'ils sont déjà disponibles.

* En TCP, cela peut se produire lorsqu'un segment est perdu et que le récepteur doit attendre sa retransmission avant de traiter les segments suivants.
* Cela impacte la latence et dégrade les performances des applications temps réel.
* Ce problème est réduit dans certains protocoles comme QUIC, qui permet un traitement plus flexible des segments reçus.

# TCP Congestion Control

TCP doit gérer la congestion dans le réseau pour éviter un effondrement global. Les principales phases et mécanismes sont :

### Congestion Avoidance

Slow start in to augmentation très agressif.

**Slow Start** :  
 Au démarrage de la connexion, l’état du réseau est inconnu. TCP commence avec une valeur de cwnd initiale (souvent 1 segment, ou une valeur plus élevée dans certaines versions récentes, jusqu’à environ 10 segments) et augmente de façon exponentielle – chaque ACK reçu incrémente cwnd de 1, doublant ainsi la fenêtre par intervalle RTT.

**Limitation** :  
 Cette augmentation rapide est agressive et peut provoquer rapidement une congestion. C’est pourquoi le seuil de démarrage (ssthresh) est utilisé pour déterminer quand passer du mode « Slow Start » au mode « Congestion Avoidance ».

Congestion avoidance

**Principe** :

Lorsque cwnd dépasse ssthresh, TCP passe en mode Congestion Avoidance pour augmenter la fenêtre plus prudemment. Pour chaque RTT, cwnd est augmenté d’environ 1 segment (précisément, cwnd = cwnd + 1/cwnd pour chaque ACK reçu).

### Fast Retransmit

Il y a un ACK timeout qui va indiquer s' il y a une perte, mais cela peut être long.Donc l’idée est d'utiliser une duplication d’Acks pour détecter la congestion. Une duplication d’ACK peut provenir de multiples raisons :

* Congestion
* Les segments utilisent différents chemins
* Reordered Acks

Donc on ne va pas considérer après la première duplication que le segment est perdu cela serait trop agressif pour le réseau, donc on va attendre 3 Acks dupliqués donc ce qui fait un total de 4 acks identiques ( 1 l’original et 3 dupliquées).

Avantages :

* Permet d’éliminer les timeouts
* 20 % de débit supplémentaires souvent
* Ne fonctionne pas quand la fenêtre de transmission est trop petite car ne permet pas la réception de plusieurs Acks dupliqués
* Si plusieurs segments sont perdus dans la même fenêtre alors ça ne peut fonctionner.

**Limites** :

* Ne fonctionne pas si la fenêtre de transmission est trop petite pour recevoir 3 ACK dupliqués.
* Si plusieurs segments sont perdus dans la même fenêtre, ce mécanisme peut s’avérer insuffisant.

### Fast Recovery

Il est en complément de [Fast Retransmit](#_urvlc9ozlyvn), la réception d’Acks dupliqués implique que le réseau fonctionne même si on a perdu un segment. Donc aller dans un Slow Start est une mauvaise idée, car la congestion a sûrement disparu, on pourrait repartir d’un peu plus qu’une fenêtre de 1.

Donc on peut entrer dans ce mode après le [Fast Retransmit](#_urvlc9ozlyvn). Comme d’habitude ssthresh = FlightSize/2, on retransmet après un segment perdu. Ce qu’on va faire c’est qu’on va ajouter

cwnd = sshtrhesh + ndup (nb d’Acks dupliqués)

Cela permet donc de transmettre de nouveaux segments et donc d’éviter le slow start et de partir directement à la [Congestion Avoidance](#_y9x832b4kxrt). On « gonfle » temporairement cwnd en le fixant à ssthresh + (ndup), où ndup est le nombre d’ACK dupliqués reçus.

**Impact :**  Il n’y a plus la montée exponentielle de 1 au Threshold après avoir atteint le max ( par exemple : timer expiration…)

### Selective Acknowledgements

Le récepteur peut ACK de façon continue, mais on a pas de ACK pour les segments qui ont bien été reçu après un segment perdu. Donc l’émetteur n’a pas de feed-back sur tous les segments qui ont bien été reçus.

Idéalement l’émetteur devrait renvoyer uniquement les segments perdus.

Avec SACK (SelectiveACK), le récepteur fournit un feed-back à l'émetteur via une extension de l’entête TCP.

**Problème sans SACK** :

Le récepteur ACK seulement les segments reçus de façon contiguë, ce qui ne permet pas de savoir quels segments ont été correctement reçus après une perte.

## TCP Mechanisms

### Delayed ACKs

Il existe beaucoup d'applications avec des communications bidirectionnelles ( HTTP, Payement bancaire …). le récepteur peut retarder l’envoi de l’ACK (jusqu’à 500 ms) pour pouvoir l’associer à des données sortantes (piggybacking, data).

**Règle** :

Un ACK doit tout de même être envoyé pour au moins tous les deux segments reçus.

On peut donc réduire “l’overhead” en combinant plusieurs ACKs dans un segment.

On fait du piggy-backing car on met l’ACK dans la data.

### 

### Nagle’s Algorithm

**Objectif** :

Éviter l’envoi de petits segments qui augmenteraient l’overhead (cas fréquent dans certaines applications comme l’IoT).

Un autre souci, ce sont les applications qui font passer dans le réseau des petits segments du à leur fonctionnalités ( par exemple : IOT …). Cela peut créer des soucis car on peut avoir une entête plus grande que la donnée elle-même. Donc on va attendre que au niveau que la couche applicative fournit plus de data afin de les combiner en un seul segment.

Tant que l’ACK n’est pas reçu, on continu bufferisé la sortie de la data jusqu’à l’obtention d’un segment rempli et donc qu’il puisse être envoyé.

**Interaction** :

Attention à son interaction avec le Delayed ACK, car cela peut introduire des délais supplémentaires.

## TCP Versions

Différentes versions implémentent différents mécanismes.

* TCP TAHOE
* TCP CUBIC
* TCP RENO
* TCP NEW RENO
* TCP VEGAS

**Major Requirement**

Cependant ces différentes versions, doivent être toutes compatibles avec les versions précédentes car cela pourrait avoir de grosses répercussions sur internet.

**Other Requirement**

TCP friendliness, TCP doit respecter une équité au flux TCP, une nouvelle version ne doit pas détruire cette équité entre les flux.

**Common Principle**

**AIMD -**  Additive Increase Multiplicative Decrease, on doit toujours augmenter la fenêtre avec addition et la baisse avec des divisions.

## TCP CUBIC

**Philosophie :** Chercher à améliorer l’approche **AIMD** en adaptant la fenêtre de congestion en fonction du temps écoulé depuis la dernière congestion.

Le niveau de congestion ne change pas beaucoup, et la congestion apparaît à cwnd=Wmax.

Et on décroit jusqu'à cwnd = Wmax/2. Puis on augmente a cwnd de manière agressive.

Donc on définit notre fenêtre comme une fonction qui dépend du temps de la dernière congestion.

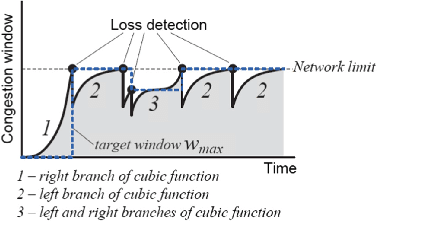
De manière local nous avons un slow start quand cwnd est proche de Wmax, la valeur de la fenêtre dépend de la dernière congestion.

Si il n’y a pas de congestion proche de Wmax, il n’y a aucun moyen de déterminer le niveau de congestion.

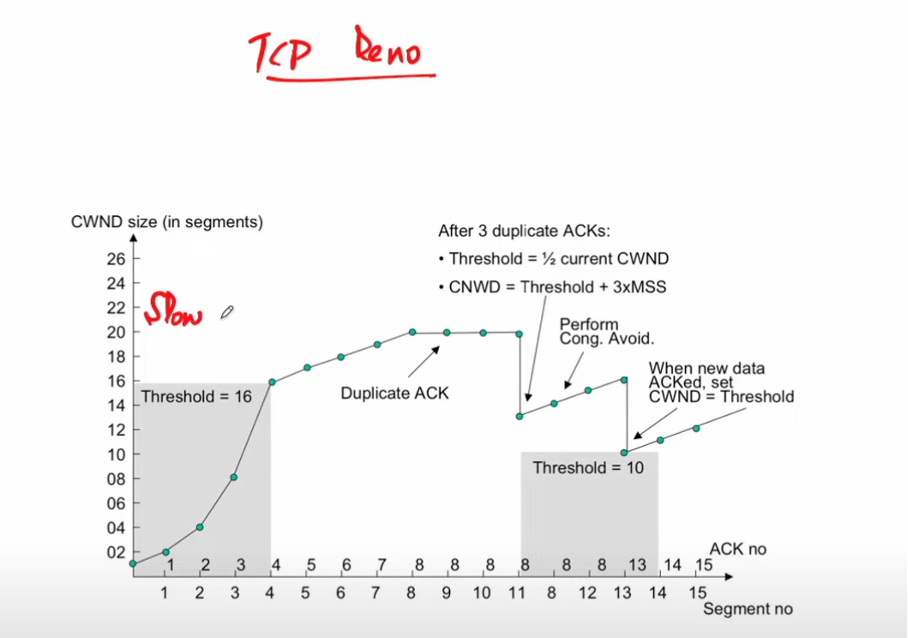
Finalement la fenêtre varie en fonction du temps et non des ACKs reçus, donc **la fenêtre cubic est une fonction de temps de la dernière congestion**.

**Principe** :

* La congestion se produit lorsque cwnd atteint Wmax.
* En cas de perte, cwnd est réduit à Wmax/2.
* La croissance de cwnd est alors déterminée par une fonction cubique du temps depuis la dernière congestion, permettant une augmentation agressive lorsque l’on est loin de Wmax, puis plus prudente à l’approche de Wmax. (Wmax est le point auquel on a subit la dernière perte de paquet



## TCP Reno



TCP Reno est une amélioration de TCP Tahoe, introduite pour mieux gérer la congestion du réseau. Il repose sur plusieurs mécanismes :

1. **Slow Start** : La fenêtre de congestion (cwnd) augmente exponentiellement au début pour rapidement atteindre la capacité du réseau.
2. **Congestion Avoidance** : Une croissance linéaire de cwnd pour éviter la congestion.
3. **Fast Retransmit** : Lorsqu’un paquet est perdu (détecté par 3 accusés de réception dupliqués), Reno retransmet immédiatement le paquet perdu au lieu d’attendre le timeout.
4. **Fast Recovery** : Après une perte, TCP Reno réduit de moitié la fenêtre de congestion et passe directement en **congestion avoidance**, évitant ainsi de recommencer en slow start.

### Limites de TCP Reno

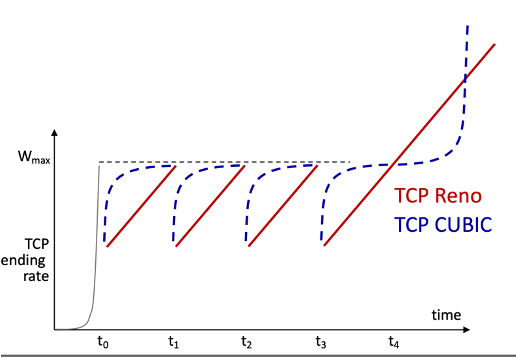
❌ **Moins efficace sur les réseaux à haut débit et forte latence**, car il réagit mal aux pertes de paquets.

❌ **Sensible aux pertes de paquets non causées par la congestion** (ex : réseaux sans fil).

### Évolution

TCP Reno a été remplacé par des algorithmes plus performants comme **TCP CUBIC** (par défaut sur Linux) et **TCP BBR**, qui offrent une meilleure gestion de la bande passante et de la latence

TCP **New Reno** est une version améliorée de **TCP Reno**, qui optimise la récupération après une perte de paquets en améliorant la phase de **Fast Recovery**. Il est plus efficace dans les environnements où plusieurs paquets sont perdus.



Cubic exploité mieux la BP que Reno.

## TCP BBR

**But** :

Stabiliser et augmenter le débit de transmission de manière plus douce, en détectant l’état du réseau à partir du RTT.

Le réseau peut être dans trois états :

* **Under-Utilized** : Le Débit est en dessous de la capacité du lien et pas de queues formées dans les routeurs
* **Over-Utilized** : Débit est au dessus de la capacité du lien et des queues sont formées dans les routeurs
* **Saturated** : Les queues sont remplies et les paquets sont perdus.

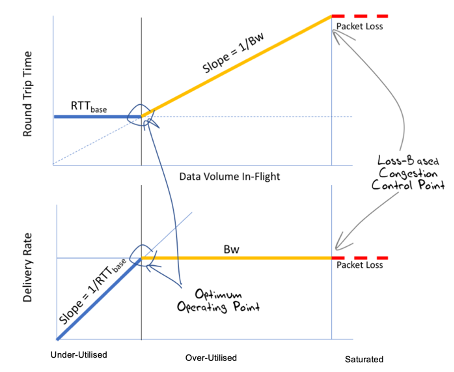
Loss-based TCP versions probe vers des états saturés et vont drop rapidement vers Under-Utilized, et à nouveau ensuite vers saturated.

Donc l’objectif de **TCP BBR** est de se stabiliser et augmenter de façon plus douce lorsqu’on arrive dans l’état Over-Utilized et on continue jusqu’au paquet perdu ( Loss-Based Congestion Control Point, le moment où le premier paquet est perdu).

**BBR -** Bottleneck Bandwith and Round-trip propagation time

**L’idée** est de détecter l’Over-Utilization en monitorant de façon douce le RTT. On va donc sonder par intermittence.

* Si le RTT est similaire au dernier RTT mesuré, alors le réseau est Under-Utilized et donc on peut augmenter le taux de transmission.
* Si le RTT augmente, alors il semble avoir une Over-Utilization.
* On cadence donc le taux de transmission pour rester proche du point optimal operation
* **NOT VERY TCP FRIENDLY**

****

**Le premier graphique donne l’évolution du RTT et le second celui de la BP.**

**BBR est surtout utilisé dans les datacenters**

## Network-Assisted Solutions

Au-delà du “end-to-end” principe, les paquets sont perdus dans le réseau, on peut aussi implémenter des techniques avancées au niveau des routeurs.

### Random Early Detection

**Principe** :  
 Les routeurs gèrent leurs files d’attente et commencent à dropper des paquets de façon probabiliste lorsque la file dépasse un certain seuil.

Donc finalement on va affecter uniquement les TCP senders qui vont entrer en Slow Start ou Congestion Avoidance, ce qui va permettre de réduire l’usage du réseau avant l'actuelle congestion.

**But** :  
 Prévenir la congestion en incitant les émetteurs TCP à réduire leur débit avant que la file ne se remplisse complètement.

### Explicit Congestion Notification

Le routeur va tag les paquets au lieu de les drop, on est à l'opposé de **RED**, ce qui permet à TCP d’apprendre l’information et réduit la fenêtre de congestion sans perte de paquets et donc réduire le délai par les retransmissions.

**Avantages** :

Permet à TCP de réduire la fenêtre de congestion sans perte de paquets et diminue ainsi les délais dus aux retransmissions.

## Transport layer security

TCP et UDP n’ont pas de fonctions de sécurité, l’idée c’est qu’elle doit être implémentée au niveau de la couche applicative. Ce n’est pas scalable avec la démocratisation des applications de développement, la sécurité est seulement nécessaire au niveau du système d’opération.

### Transport Layer Security (TLS)

* **Objectif** :  
   Fournir l’authentification et le chiffrement sur une connexion TCP.

**Mécanismes d’établissement** :

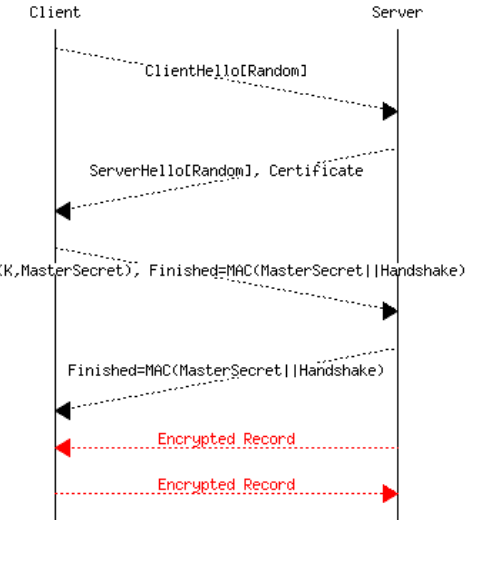
* **ClientHello** : Le client envoie la version du protocole, un nombre aléatoire (pour la génération des clés), les suites de chiffrement et d’éventuelles extensions (ex. SNI).
* **ServerHello** : Le serveur répond avec sa version, un nombre aléatoire, un Session ID et d’autres paramètres.
* **Certificate** : Permet l’authentification du serveur via une infrastructure à clé publique (requiert une Autorité de Certification).
* **MasterSecret** : Établi via un échange de clés, il est ensuite utilisé pour dériver les clés de session.
* **Finished** : Message d’authentification qui assure que le processus n’a pas été modifié.

Utilisation du protocole TLS :

* Evolution du SSL
  + Responsable de l’authentification et d’encryptage
* Au dessus d'une connexion faible TCP

Deux implémentations possible :

* Utilisation d’un port différent pour une connexion sécurisé :
  + HTTP port 80 ; HTTPS port 443
* On utilise le TCP port et un message spécifique pour commencer la connexion :
  + STARTTLS via SMTP



## TCP Implementation

**Au niveau du système d’exploitation** :

TCP (et UDP) sont intégrés directement dans le noyau via une série de bibliothèques. Les applications y accèdent via l’API Socket (les sockets apparaissent comme des descripteurs de fichiers sous Unix).

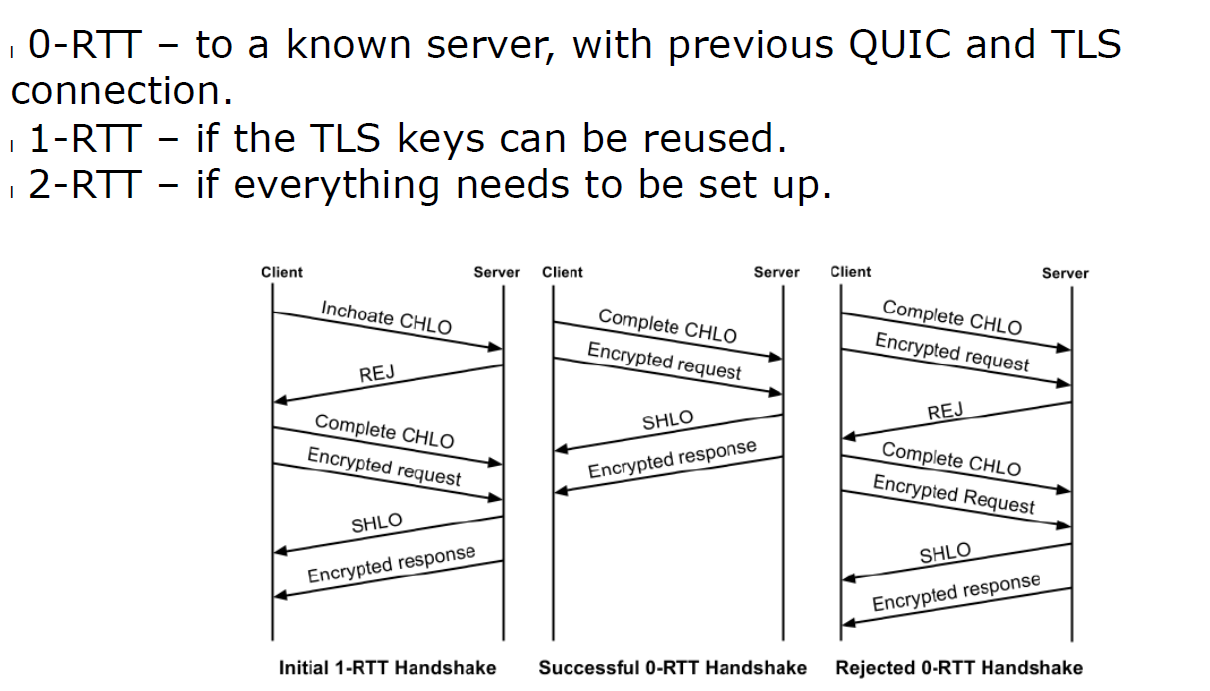
**Internet Ossification (non évolution d’internet, du au grand nombre de filtres mis en place dans le réseau)** :  
 La présence de nombreux intermédiaires (firewalls, NAT, proxies) rend difficile l’évolution des protocoles de transport, d’où le développement de solutions comme QUIC.

## QUIC

**Contexte et Motivation** :  
 Conçu par Google pour répondre aux problèmes d’ossification d’Internet, QUIC intègre TLS directement et fonctionne sur UDP pour contourner les filtres des intermédiaires.

**Caractéristiques** :

* Utilise un identifiant de connexion (Connection ID) permettant de migrer la connexion entre différentes adresses IP et interfaces.
* Gère plusieurs flux indépendamment pour éviter le Head-of-line blocking, améliorant ainsi la réactivité par rapport à TCP.
* Réduit le délai de connexion avec des handshakes 0-RTT, 1-RTT ou 2-RTT selon la reprise de session.
* Utilise un unique identifiant pour le segment ce qui facilite la retransmission et le calcul du RTT



**Adoption** :  
 Utilisé par de grands acteurs comme Google, Facebook et Cloudflare, QUIC permet de réduire le temps de recherche (search time) et le rebuffering vidéo.

TD

## Exercice 1:

definition du SRTT : Le **SRTT** (**Smoothed Round Trip Time**) est une estimation du **temps aller-retour moyen** d’un paquet dans TCP. Il est utilisé pour ajuster **le délai d’attente avant une retransmission** et améliorer les performances du protocole.

### 1) Que se passe-t-il si le RTT est largement sous-estimé?

Avoir une valeur largement sous-estimée du RTT aboutit à avoir une valeur très

petite du timeout du timer déclenché pour l'attente de l'acquittement d'un

segment envoyé. Ainsi, on va détecter que des congestions ont lieu sur le réseau

alors que ce n'est pas le cas et on va retransmettre des segments bien reçus.

2) Quelles sont les conséquences d'une sur-estimation?

Une surestimation aboutit à avoir une grande valeur du timeout du timer

déclenché pour l'attente d'acquittement du segment envoyé. Ainsi, si un segment

est perdu, on va attendre pour une longue période avant de le retransmettre, ce

qui pourrait affecter le débit utile du réseau, surtout si on recommence avec une

phase de slow start.

### 3) SRTT(k)=αSRTT(k−1)+(1−α)RTT(k−1)

En suivant l'hypothèse que les 20 segments sont reçus avec le même RTT, on peut

remarquer qu'il s'agit d'une suite arithmético-géométrique de la forme: Un+1=aUn+b

Comme a≠1 , en posant r= b/(1-a) on peut deduire: Un=an

(U0−r)+r

Dans notre cas, r=1 , d'où: Un=αn

(U0−1)+1

Pour α=0.9, on aura U20=1.3647 s

Pour α=0.6, on aura U20=1.0001 s

### 4) Quelles sont vos conclusions?

Si le réseau est stable, on a intérêt d'utiliser une petite valeur de α . Une grande valeur a

plus de sens dans le cas d'un réseau instable, qui donne plus de poids au dernier SRTT mesuré.

## Exercice 2:

## 

### 1) Quelle est la taille de fenêtre (constante) annoncée par le récepteur?

La taille de la fenêtre annoncée par le récepteur vaut 20. On remarque dans la figure 2

que l'émetteur transmet d'un seul coup 20 segments avant d'attendre quelques instants

pour envoyer le segment numéro 21. La durée attendue entre l'émission du segment 20

et le segment 21 correspond au temps d'attente de l'acquittement du segment 1 (RTT).

### 2) Donnez le contenu de la fenêtre d'émission en chacun des points marqués sur l'axe

des temps.

Rappel: Le contenu de la fenêtre d'émission correspond aux segments que l'émetteur a

droit de transmettre. Sa taille et son contenu sont définis en fonction des mécanismes

de TCP implémentés.

Le temps qu’on a entre les émissions des paquets en diagonales qui correpondent a un ack reçu correspond au temps RTT.

**Point A:**

Contenu de la fenêtre d'émission: Seg 0 → Seg 19 Attention : Il n'envoie au début que

Seg 0 pour vérifier le bon fonctionnement du réseau!

Le temps entre le point A et le premier paquet émis etst le temps de remplissage du bffer de l’emetteur.

**Point B:**

Le temps entre le premier paquet émis et le point B correspond au temps pour vérifier la communication de bout en bout.

L'émetteur a reçu l'acquittement du segment 0. Contenu de la fenêtre d'émission: Seg 1

→ Seg 20

**Point C:**

L'emetteur a reçu les acquittements des segments: Seg 1 → Seg 15 Il attend l'

acquittement du segment Seg 16 qui est en fait détruit. Contenu de la fenêtre d'émission:

Seg 16 →Seg 35

On est en attente du délai de réémission du paquet perdu.

**Point D:**

Entre le point B et D nous avons un temps SRTT = 1,6 s

Le time-out du timer lancé en attente de l'acquittement du Seg 16 a déjà expiré.

Le Seg 16 a été retransmis. On reçoit son acquittement et on envoie le Seg 17. Contenu de

la fenêtre d'émission: Seg 17 → Seg 36

**Point E:**

A ce moment là, tous les segments avant le segment 36 sont déjà acquittés. Contenu de la

fenêtre d'émission: Seg 36 → 55

On comprend qu'avec la destruction des paquets que la capacité du routeur est de 15 car on perd toujours un segment au bout du 15 ème.

### 3) Quelle est la capacité du routeur?

Le routeur est capable de garder 15 segments dans sa mémoire. On peut déduire

cela au moment de la première émission de 20 segments successifs, dont les cinqs

derniers sont rejetés. Le suivi de l'évolution des segments détruits à travers le

temps le confirme aussi.

### 4) Quelle est, approximativement, la durée du délai de réémission?

La première réémission a eu lieu avec un délai de 1.5 s. La deuxième réémission a

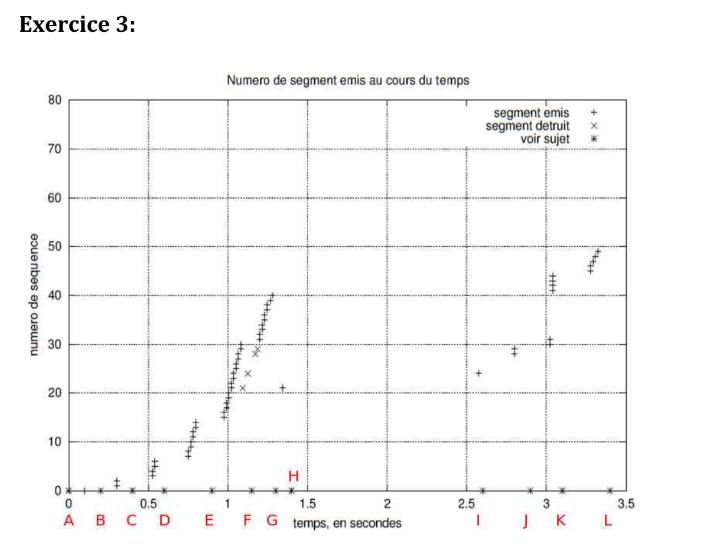
eu lieu avec un délai de 1 s. Alors qu'on aurait dû s'attendre à des délais plus

similaires, cette différence est dûe à la variation de la durée du time-out du timer

d'attente d'acquittement qui lui-même dépend de la valeur estimée du RTT et qui

varie avec le temps.

## Exercice 3



1) Expliquez les différentes phases se déroulant dans les 3.5 premières secondes.

2) Donnez, en chacun des points marqués d'un astérisque sur l'axe des temps: le

contenu de la fenêtre d'émission, la taille de la fenêtre de congestion, et la valeur du

seuil ssthresh.

3) Expliquez en quoi les deux premières réémissions de segments perdus diffèrent.

Dans ce qui suit, on utilise les symboles suivants:

- swnd\_content: Contenu de la fenêtre d'émission

- cwnd\_size: La taille de la fenêtre de congestion

- ssthresh: la valeur du seuil ssthresh

**Point A:**

- swnd\_content: Seg 0

- cwnd\_size: 1

- ssthresh: rnd\_high

On est tout au début de la phase de slow-start avant d'envoyer tout segment.

ssthresh prend une valeur initiale aléatoire généralement très grande rnd\_high.

**Point B:**

- swnd\_content: Seg 0

- cwnd\_size: 1

- ssthresh: rnd\_high

On n'a pas encore reçu l'acquittement du Seg 0.

**Point C:**

- swnd\_content: Seg 1 → Seg 2

- cwnd\_size: 2

- ssthresh: rnd\_high

On attend les acquittements des segments Seg 1 et Seg 2.

**Point D:**

- swnd\_content: Seg 3 → Seg 6

- cwnd\_size: 4

- ssthresh: rnd\_high

On attend les 'acquittements des segments Seg 3 → Seg 6.

**Point E:**

- swnd\_content: Seg 7 → Seg 14

- cwnd\_size: 8

- ssthresh: rnd\_high

On attend les acquittements des segments Seg 7 → Seg 14.

**Point F:**

- swnd\_content: Seg 15 → Seg 30

- cwnd\_size: 16

- ssthresh: rnd\_high

On attend les acquittements des segments Seg 15 → Seg 30.

**Point G:**

- swnd\_content: Seg 21

- cwnd\_size: 1

- ssthresh: 5 (visible uniquement à partir du point L)

L'émetteur reçoit trois acquittements pour Seg 20, avec fast retransmit on détecte la

congestion et on entre en slow start.

**Point H:**

- swnd\_content: Seg 24

- cwnd\_size: 1

- ssthresh: 5

Le récepteur indique que le dernier segment reçu est Seg 23.

**Point I:**

- swnd\_content: Seg 24

- cwnd\_size: 1

- ssthresh: 5

On détecte que le Seg 24 est détruit à cause de l'expiration du timer, on entre de

nouveau en slow-start et on retransmet le segment. On n'a pas encore son acquittement.

**Point J:**

- swnd\_content: Seg 29 → Seg 31

- cwnd\_size: 2

- ssthresh: 5

Le récepteur a reçu le segment Seg 28, mais pas encore le segment Seg 29.

**Point K:**

- swnd\_content: Seg 41 → Seg 44

- cwnd\_size: 4

- ssthresh: 5

Le récepteur a reçu tous les segments jusqu'à Seg 40. L'émetteur lui envoie les Seg 41 →

Seg 44.

TP 1

### Sockets TCP sous Linux

#### **1️⃣ Création d’une socket**

int serveur\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

* AF\_INET → IPv4
* SOCK\_STREAM → TCP
* Retourne un descripteur de fichier (>2, car 0=stdin, 1=stdout, 2=stderr). ;Si la socket est la première ouverte dans le programme, le descripteur pourrait être **3**.
* **Un nombre positif** : Cela représente le descripteur de fichier de la socket créée. Il s'agit d'un identifiant unique pour la socket dans le système d'exploitation.

📌 **Toujours vérifier les erreurs :**

if (serveur\_socket == -1) {

perror("Erreur socket");

exit(EXIT\_FAILURE);

}

#### **2️⃣ Réutilisation d’un port (SO\_REUSEADDR)**

Permet d'éviter les erreurs si le programme est redémarré rapidement.

int reuse = 1;

setsockopt(serveur\_socket, SOL\_SOCKET, SO\_REUSEADDR, &reuse, sizeof(reuse));

#### **3️⃣ Configuration de l’adresse (struct sockaddr\_in)**

* INADDR\_ANY → Accepte toutes les connexions.
* htons(port) → Convertit le port au format réseau (Big Endian).

struct sockaddr\_in my\_addr;

memset(&my\_addr, 0, sizeof(my\_addr));

my\_addr.sin\_family = AF\_INET;

my\_addr.sin\_port = htons(8080);

my\_addr.sin\_addr.s\_addr = INADDR\_ANY;

#### **4️⃣ Liaison de la socket à une adresse et un port**

if (bind(serveur\_socket, (struct sockaddr\*)&my\_addr, sizeof(my\_addr)) < 0) {

perror("Erreur bind");

close(serveur\_socket);

exit(EXIT\_FAILURE);

}

🚨 **Erreur courante :** "Address already in use" → Port déjà utilisé.

#### **5️⃣ Fermeture de la socket**

close(serveur\_socket);

### Gestion des connexions et communication

#### **Acceptation d’un client**

#### int client\_socket = accept(serveur\_socket, (struct sockaddr\*)&client\_addr, &client\_len);

if (client\_socket < 0) {

perror("Erreur accept");

exit(EXIT\_FAILURE);

}

* accept() crée un **nouveau** descripteur pour communiquer avec le client.
* En cas d’échec, retourne -1.

#### **Communication avec recv() et send()**

**Envoi :**

send(client\_socket, "Hello", 5, 0);

**Réception :**

char buffer[1024];

int bytes\_received = recv(client\_socket, buffer, sizeof(buffer), 0);

if (bytes\_received > 0) {

buffer[bytes\_received] = '\0'; // Assurer une terminaison correcte

}

📌 **Si trop de données envoyées :**

* Seuls sizeof(buffer) octets sont reçus.
* Les autres restent en attente pour la lecture suivante.
* une **troncature** peut se produire. Les données excédentaires risquent d'être perdues, car le buffer ne pourra pas les stocker.

📌 **Si pas assez de données :**

* recv() lit ce qui est disponible sans toucher au reste du buffer.

### Conversions de format réseau

* htons() → Convertit un **port** en **Big Endian**.
* htonl() → Convertit une **adresse IP** en **Big Endian**.
* ntohs() et ntohl() → Conversion inverse (réseau → machine).

Exemple (client) :

struct sockaddr\_in serveur\_addr;

serveur\_addr.sin\_port = htons(8080);

inet\_aton("127.0.0.1", &serveur\_addr.sin\_addr);

Gestion des processus (fork)

* fork() retourne :  
  + **0** → Code exécuté par le fils.
  + **PID du fils** → Dans le père.
  + **-1** → Erreur.

Exemple :

pid\_t pid = fork();

if (pid == 0) {

// Code exécuté par le fils

} else if (pid > 0) {

// Code exécuté par le père

} else {

perror("Erreur fork");

}

### fork() et la gestion des processus

#### **Définition**

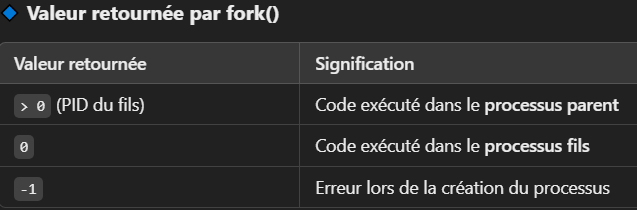
fork() crée un **nouveau processus** (processus fils) qui est une copie presque exacte du processus parent.

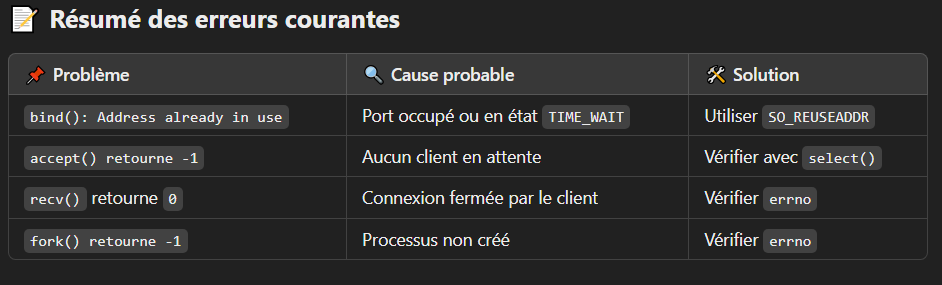
#### **Utilisation dans un serveur TCP**

Lorsqu’un **serveur TCP** reçoit une connexion, il peut utiliser fork() pour traiter chaque client dans un processus fils :

✅ **Avantages** :

* Permet au serveur de gérer plusieurs clients simultanément.
* Le **père** continue à écouter de nouvelles connexions pendant que le **fils** gère le client.





# TP2

| Version TCP | Trafic en compétition | Capacité liens  (Mbps) | Drop de paquets | Pertes routeur | Débit obtenu  (Mbps) | Observation |
| --- | --- | --- | --- | --- | --- | --- |
| Cubic | non | 10 | 0 | 0 | 11,5 |  |
| Cubic | non | 1,024 | 0 | 42 | 1,08 |  |
| Cubic | oui | 10 | 0 | 0 / 0 | 3,41 / 10,1 |  |
| Cubic | oui | 1,024 | 0 | 46 / 56 | 0,63 / 0,558 |  |
| Reno | non | 10 | 0 | 0 | 9,83 |  |
| Reno | non | 1,024 | 0 | 235 | 0,987 |  |
| Reno | oui | 10 | 0 | 2 / 780 | 0,716 / 9,24 |  |
| Reno | oui | 1,024 | 0 | 303 / 311 | 0,511 / 0,498 |  |
| BBR | non | 10 |  |  |  |  |
| BBR | non | 1,024 | 0 | 60 | 1,05 |  |
| BBR | oui |  |  |  |  |  |
| BBR | oui | 1,024 | 0 | 302 / 311 | 0,510 / 0,510 |  |
| Cubic / Reno | oui | 10 | 0 | 8 / 843 | 2,76 / 10,5 |  |
| Cubic / Reno | oui | 1,024 | 0 | 60 / 90 | 0,601 / 0,552 |  |
| Cubic / Reno | oui | 1,024 | 0 | 218 / 328 | 0,525 / 0,482 |  |
| Cubic / BBR | oui | 10 | 0 | 0 / 0 | 10,3 / 1,18 |  |
| Cubic / BBR | oui | 1,024 | 0 | 74 / 108 | 0,553 / 0,544 |  |
| Reno / BBR | oui | 1,024 | 0 | 101 / 112 | 0,378 / 0,822 |  |
| Reno / BBR | oui | 10 | 0 | 0 / 0 | 8,44 / 1,89 |  |
| Reno / BBR | oui | 1,024 | 0,1 | 99 / 176 | 0,288 / 0,75 |  |
| Reno / BBR | oui | 10 | 0,1 | 285 / 994 | 2,38 / 7,17 |  |
| Reno / Cubic | oui | 10 | 0,1 | 385 / 334 | 2,31 / 2,25 |  |
| Reno / Cubic | oui | 1,024 | 0,1 | 122 / 116 | 0,611 / 0,506 |  |
| BBR / Cubic | oui | 1,024 | 0,1 | 188 / 100 | 0,718 / 0,331 |  |
| BBR / Cubic | oui | 10 | 0,1 | 1018 / 275 | 7,26 / 1,86 |  |

# DS

### Exercice 1 : New Reno vs. Cubic (8 points) Considérons deux clients TCP (nommés A et B) connectés à un serveur S et les hypothèses suivantes :

- le RTT entre A et S est de 50 ms ;

- le RTT entre B et S est de 150 ms ;

- les deux clients ont 25 segments chacun à transmettre au serveur ;

- pas de perte de segment ou d'ACK.

Si nous considérons que les deux clients et le serveur utilisent la version TCP New Reno (slow start + congestion avoidance + fast retransmit + fast recovery) :

1.1. En utilisant des chronogrammes, montrez l'évolution de la fenêtre de congestion de A et de B.

| RTT | A/tot | B |
| --- | --- | --- |
| 1 | 1 | 1 |
| 2 | 2/ 3 | 2 |
| 3 | 4 /7 | 4 |
| 4 | 8 /15 | 8 |
| 5 | 16 /31 (les 25 paquets sont envoyées) | 16 |

Donc il faut 5 RTT pour envoyer les 25 paquets étant donné, que A et B n’ont pas le même RTT ils n’enverront pas les 25 paquets a la même vitesse.

1.2. Faites une estimation des débits des deux clients (pas besoin d'être ultra-précis).

Rb = data/tps

On a pour A :

T = 5\*50ms = 250ms

Data = 1460octets \*25 \* =36.5 Ko \* 8 (bits) = X K

D’où Rb\_A = 146 Kops

Pour B :

T\_b = 5\*150ms = 750ms

Rb\_B = 48 Ko/s \* 8 (bits) = X Kb/s

Supposons maintenant que les clients et le serveur utilisent une version de TCP Cubic.

1.3. Expliquez comment évoluent les fenêtres de congestion dans ce cas.

TCP **Cubic** est conçu pour améliorer l'équité entre les flux ayant des RTT différents et pour mieux utiliser les liens à haut débit. Contrairement à TCP **New Reno**, qui augmente la fenêtre de congestion de manière linéaire en congestion avoidance, TCP Cubic utilise une fonction **cubique** pour contrôler l'évolution de la fenêtre de congestion.

La **croissance de cwnd** suit trois phases :

1. **Croissance rapide** après une réduction due à la congestion.
2. **Croissance plus lente** lorsqu'elle s'approche de WmaxW\_{\text{max}}Wmax​.
3. **Croissance rapide après WmaxW\_{\text{max}}Wmax​** si aucune perte n’est détectée.

TCP Cubic favorise les connexions avec un **RTT plus long**, réduisant ainsi l'avantage que TCP New Reno donnait aux flux avec un RTT plus court.

**Effet sur A (RTT = 50ms) et B (RTT = 150ms)** :

* Avec TCP New Reno, **A a un énorme avantage** car il augmente cwnd plus rapidement.
* Avec TCP Cubic, **B bénéficie d'une croissance plus rapide dans la phase post-congestion**, ce qui l'aide à compenser son RTT élevé.

La différence de débit entre A et B sera **réduite** par rapport à New Reno, mais A gardera un débit supérieur en raison de son RTT plus faible.

1.4. Est-ce que le serveur a toujours besoin d'envoyer des acquittements aux clients en TCP Cubic ? Pourquoi ?

Oui, **le serveur doit toujours envoyer des acquittements (ACK) aux clients en TCP Cubic**, comme dans toute autre variante de TCP.

TCP Cubic ajuste sa fenêtre de congestion (**cwnd**) en fonction de la réception des ACKs.

Chaque ACK reçu permet au client d’augmenter progressivement sa **cwnd** (notamment pendant **slow start** et **congestion avoidance**).

En cas de perte détectée (absence de plusieurs ACKs), TCP Cubic réduit **cwnd** et ajuste son algorithme cubique.

TCP mesure le **Round-Trip Time (RTT)** en utilisant les ACKs pour ajuster ses délais de retransmission (**RTO - Retransmission Timeout**).

Cette mesure est essentielle pour adapter la croissance de **cwnd** en TCP Cubic.

TCP CUBIC utilise aussi fast retransmit qui est basé sur les ack reçues

1.5. Comment se passe la cohabitation entre des flux TCP New Reno et TCP Cubic ?

Cela peut entraîner des **interactions complexes** et, parfois, des **problèmes d'équité** entre ces flux.

#### **Effet de la cohabitation sur les flux TCP :**

* **Interaction entre les algorithmes de contrôle de congestion :**
  + **TCP New Reno** réagit plus lentement à la congestion car sa croissance est linéaire.
  + **TCP Cubic** a une croissance plus rapide après une perte de segment, ce qui peut lui permettre de récupérer plus rapidement et de croître plus vite lorsqu'il est loin du seuil de congestion.
* **Problèmes d’équité :**
  + TCP Cubic tend à avoir un **avantage** sur TCP New Reno, particulièrement sur les **liens à haut débit et à long délai**. En effet, TCP Cubic peut se permettre de croître plus rapidement après une congestion, tandis que TCP New Reno reste limité par son comportement linéaire.
  + Cela peut entraîner une **injustice** où les flux utilisant TCP Cubic prennent plus de bande passante au détriment des flux New Reno, en particulier si le réseau devient congestionné.

#### **Comportement en présence de congestion :**

* **Lors d’une perte de segment :**
  + **TCP New Reno** va réduire sa fenêtre de congestion de manière significative (généralement de moitié) et repartir avec une croissance linéaire.
  + **TCP Cubic** réduit aussi sa fenêtre après une perte, mais **récupère plus rapidement** grâce à la croissance cubique et continue à augmenter sa fenêtre de manière agressive.
* Cependant, l'**équilibre** entre les deux flux dépendra de **l'ordre des pertes**. Par exemple :  
  + Si un segment envoyé par un flux TCP Cubic est perdu avant un segment envoyé par TCP New Reno, TCP Cubic va probablement récupérer plus rapidement et continuer à croître, tandis que TCP New Reno restera bloqué avec une croissance linéaire.
  + Si un segment TCP New Reno est perdu, sa fenêtre de congestion sera réduite plus rapidement, mais il reviendra ensuite à une croissance linéaire, ce qui pourrait ralentir le flux global sur le réseau.

### On donne le code suivant, d’un serveur TCP :

...

1 int desc= socket(AF\_INET, SOCK\_STREAM, 0);

2 adresse.sin\_family= AF\_INET;

3 adresse.sin\_port= htons(1234);

4 adresse.sin\_addr.s\_addr= htonl(INADDR\_ANY);

5 bind(desc, (struct sockaddr\*) &adresse, sizeof(adresse));

6 listen(desc, 5)

7 int rqst;

8 while (1) {

9 int cliLen= sizeof(client);

10 rqst= accept(desc, (struct sockaddr\*)&client, &cliLen);

11 int msgSize= recv(rqst,buffer,RCVSIZE,0);

12 send(rqst,buffer,msgSize,0);

13 }

2023-2024 Page 1

INSA Lyon 4TC – TCP Dép. TC

...

Une connexion TCP démarre avec l'échange des messages SYN, SYN-ACK, ACK. Cependant,

la transmission et la réception de ces messages ne sont pas explicites dans l'API Socket.

2.1 Dans le code ci-dessus, à quel moment (ligne de code) le serveur reçoit le message SYN envoyé par un client ?

Le serveur reçoit le message SYN lorsqu’un client tente d’établir une connexion. Cela se produit avant que accept() ne soit appelé, mais c’est à l’appel de accept() (ligne 10) que le serveur commence réellement à traiter la connexion.

2.2 Dans le code ci-dessus, à quel moment (ligne de code) le serveur transmet le message SYN-ACK au client ?

Le message SYN-ACK est envoyé automatiquement par le système en réponse au SYN du client. Cet envoi se produit juste après la réception du SYN, donc avant que accept() ne retourne. C’est toujours géré par l’OS au moment de accept() (ligne 10).

2.3 Dans le code ci-dessus, à quel moment (ligne de code) le serveur reçoit le message ACK envoyé par un client ?

Le serveur reçoit le message ACK (troisième étape du handshake TCP) juste après avoir envoyé le SYN-ACK. Ce message permet de finaliser la connexion. La réception de cet ACK se produit avant que accept() ne retourne, ce qui signifie que lorsque accept() retourne une valeur valide, la connexion est complètement établie. Donc, la réception du ACK se fait toujours implicitement dans accept() (ligne 10).

Des questions pièges mdrrrr

### Exercice 3 : Réseau routier (6 points)

La congestion est un problème dans tout type de réseau, pas uniquement dans un réseau informatique. Pourtant, dans la plupart des autres réseaux, la congestion est gérée à l’intérieur du réseau. Dans le cas d’un réseau informatique, cela supposerait d’utiliser une autre couche réseau, différente de l’approche best effort du protocole IP.

3.1. En partant du fonctionnement du réseau routier, transposez quelques mécanismes réseau (pas plus de 3) pour le contrôle de congestion dans le monde des réseaux informatiques. Expliquez comment ces mécanismes pourraient être implantés dans un réseau informatique et leurs conséquences éventuelles.

### 1. Limitation de vitesse (Contrôle de débit)

**Dans les réseaux routiers :** Les limitations de vitesse réduisent les risques d’embouteillage en évitant qu'un trop grand nombre de véhicules se retrouvent au même endroit en même temps.

**Transposition aux réseaux informatiques :** Un mécanisme similaire est le **contrôle de débit** (rate limiting), où un routeur ou un serveur limite le nombre de paquets envoyés par un expéditeur en fonction de la congestion détectée. TCP utilise déjà un tel principe via le **contrôle de congestion TCP**, notamment avec l’algorithme AIMD (Additive Increase, Multiplicative Decrease).

**Conséquences :**

* Améliore la stabilité du réseau en évitant les pics de trafic.
* Peut ralentir certaines connexions, surtout celles ayant un besoin urgent de bande passante.

### 2. Péages et voies prioritaires (Différenciation de services)

**Dans les réseaux routiers :** Certains axes routiers mettent en place des péages ou des voies réservées pour les véhicules prioritaires (bus, ambulances, véhicules de covoiturage) afin de fluidifier le trafic.

**Transposition aux réseaux informatiques :** L’équivalent serait la **qualité de service (QoS)**, qui attribue des priorités à certains types de trafic (par exemple, la voix sur IP ou la vidéo en streaming) en utilisant des mécanismes comme **DiffServ (Differentiated Services)**.

**Conséquences :**

* Améliore l’expérience des services critiques en leur garantissant une bande passante suffisante.
* Peut entraîner une inégalité d’accès pour les utilisateurs ayant un trafic non prioritaire.

### 3. Feux de signalisation et rampes d’accès régulées (Contrôle d’admission)

**Dans les réseaux routiers :** Les feux de signalisation et les rampes d’accès régulées limitent le nombre de véhicules entrant sur une route encombrée, réduisant ainsi les embouteillages.

**Transposition aux réseaux informatiques :** Un mécanisme similaire serait le **contrôle d’admission**, où un routeur ou un serveur décide d’accepter ou de rejeter une nouvelle connexion en fonction de l’état actuel du réseau.

**Conséquences :**

* Évite la surcharge du réseau en empêchant trop de connexions simultanées.
* Peut poser problème pour les nouvelles connexions, qui doivent attendre avant d’accéder au réseau.

Dans le contexte des réseaux informatiques, une meilleure approche serait d’**adapter dynamiquement** le contrôle d’admission en fonction de la congestion réelle du réseau, un peu comme les feux intelligents qui analysent le trafic en temps réel.

Par exemple :

* **Algorithmes comme RED (Random Early Detection)** permettent de **détecter** les débuts de congestion et d’anticiper en ralentissant certains flux plutôt que d’attendre un engorgement total.
* **Contrôle d’admission adaptatif** : Plutôt qu’un refus strict d’accès, un réseau pourrait **ajuster dynamiquement les priorités** ou proposer une qualité de service dégradée pour éviter d’avoir des périodes de sous-utilisation.

En résumé, si les "feux" sont purement cycliques et fixes, cela peut créer une inefficacité. Mais avec un **contrôle dynamique basé sur la charge réelle**, on peut mieux équilibrer l’usage du réseau tout en évitant la congestion.