U.E. ARES
Architecture des Réseaux

Cours 3/6 : Couche transport

Olivier Fourmaux
(olivier.fourmaux@upmc.fr)

Version 5.4

Couche transport

Compréhension des principes de base de la couche transport[1]

- multiplexage
- transfert fiable
- contrôle de flux
- contrôle de congestion

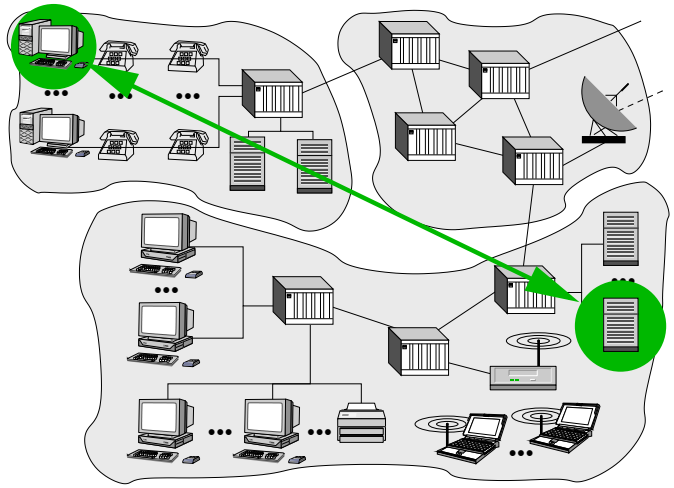
Etude des protocoles de transport dans l'Internet

- UDP : transport sans connexion
- TCP : transport orienté-connexion
- contrôle de congestion de TCP

[1] Nombreuses adaptations des slides, des schémas et du livre de J. F. Kurose et K. W. Ross, *Computer Networking : A Top Down Approach Featuring the Internet*, 3e edition chez Addisson-Wesley, juillet 2004.

Plan

Couche transport



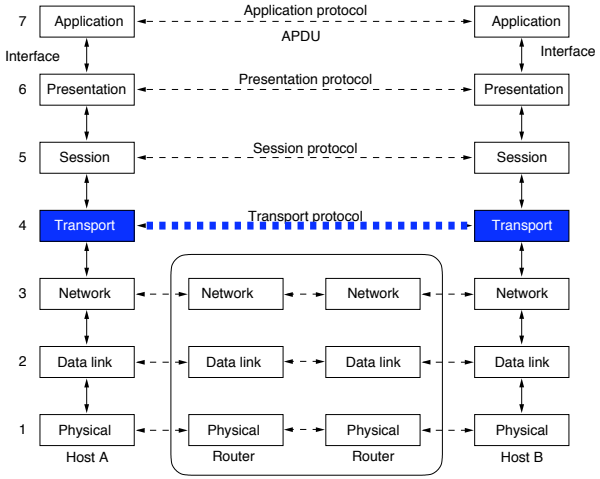
Couche transport

La **Couche transport** permet de faire **communiquer directement** deux ou plusieurs entités sans avoir à se préoccuper des différents éléments de réseaux traversés :

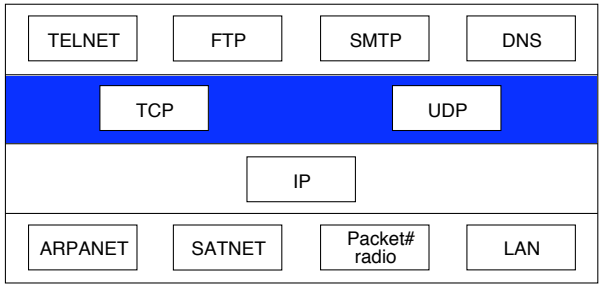
- associations virtuelles entre **processus**
- communication de bout-en-bout (*end-to-end*)
  - ✓ abstraction de la **topologie** et des **technologies** sous-jacentes
  - ✓ fonctionne dans les machines d'extrémité
    - ☞ **émetteur** : découpe les messages de la couche applicative en segments et les "descend" à la couche réseau
    - ☞ **récepteur** : réassemble les segments en messages et les "remonte" à la couche application

➠ 2 **modèles** définissent les fonctionnalités associés à chaque couche...

Couche transport : OSI



Couche transport : TCP/IP



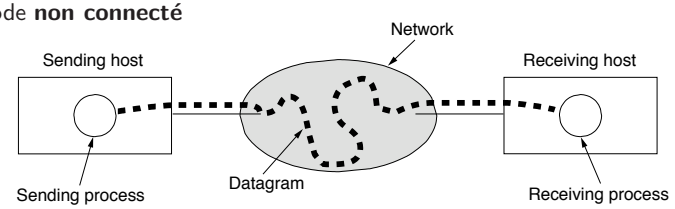
Couche transport : Internet

2 protocoles de transport standard : TCP et UDP

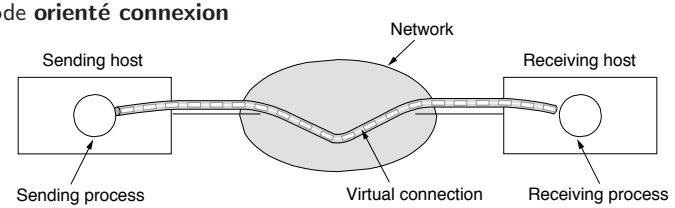
- transfert fiable et ordonné : **TCP**
  - ✓ gestion de la connexion
  - ✓ contrôle de flux
  - ✓ contrôle de congestion
- transfert non fiable non ordonné : **UDP**
  - ✓ service *best effort* ("au mieux") d'IP
  - ✓ très léger
- non disponible :
  - ✓ garanties de débit
  - ✓ garanties temporelles
    - ☞ délais non bornés
    - ☞ jigue imprévisible

Couche transport : 2 approches

Mode **non connecté**



Mode **orienté connexion**

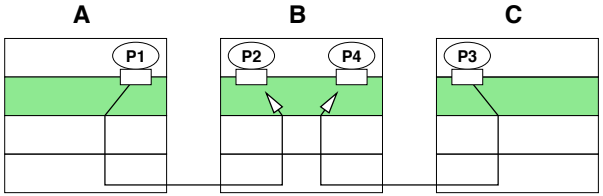


Plan

Multiplexage/Démultiplexage

Les **processus** applicatifs transmettent leurs données au système à travers des **sockets** : Le **multiplexage** consiste à regrouper ces données.

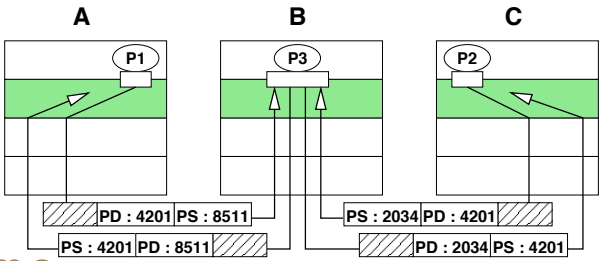
- **mux** (à l'émetteur) :
  - ✓ ajout d'un entête à chaque bloc de données d'un socket
  - ✓ collecte les données de plusieurs socket
- **demux** (au récepteur) :
  - ✓ fourniture des données au socket correspondant



Démultiplexage en mode non connecté

Association d'un socket avec un numéro de port

- identification du `DatagramSocket` : (`@IPdest, numPortDest`)
- réception d'un **datagramme** à un hôte :
  - ✓ vérification du `numPortDest` contenu
  - ✓ envoi au socket correspondant à `numPortDest`
    - ☞ ∀ `@IPsource`, ∀ `numPortSource`



Multiplexage en mode orienté connection

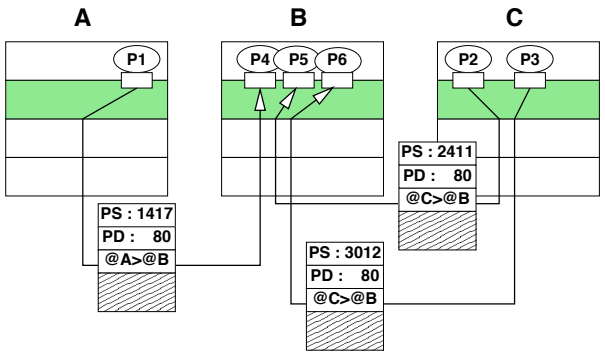
Association relative à une **connexion** entre deux processus

- identification du `StreamSocket` par le quadruplet :
  - ✓ adresse source : `@IPsource`
  - ✓ port source : `numPortSource`
  - ✓ adresse destination : `@IPdest`
  - ✓ port destination : `numPortDest`
- réception d'un **segment** à un hôte :
  - ✓ vérification du quadruplet contenu
  - ✓ envoi au socket correspondant au quadruplet
    - ☞ un serveur web peut avoir plusieurs connexions simultanée

Démultiplexage en mode orienté connection (1)

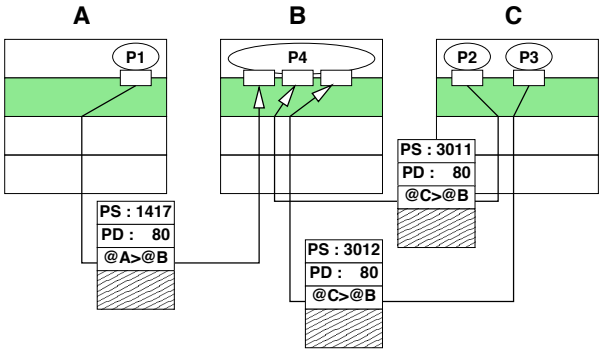
Serveur web classique (apache 1.x)

- un socket par connexion
  - ✓ HTTP en mode non persistant : un socket par requête !

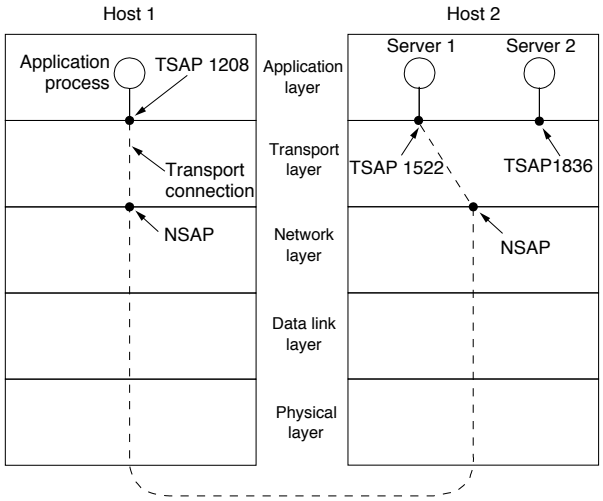


Démultiplexage en mode orienté connection (2)

Serveur web multi-threadé (apache 2.x)



Multiplexage : dénominations OSI



Plan

Rappels sur la couche transport

Multiplexage et démultiplexage

UDP : un protocole en mode non connecté
- format du datagramme UDP
- utilisation d'UDP

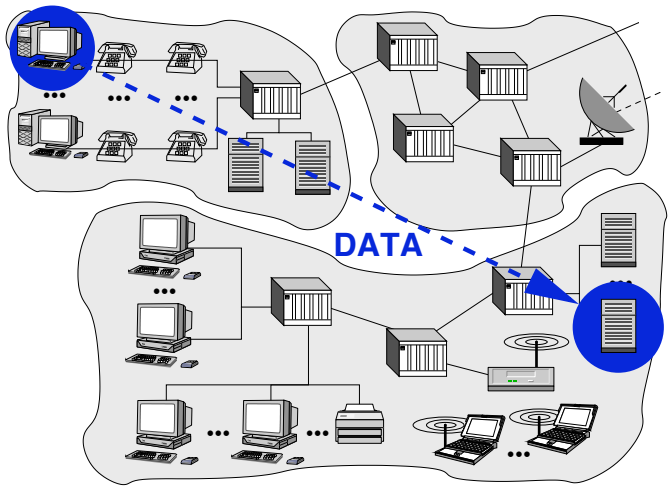
Principes de transfert de données fiable

TCP : un protocole en mode orienté connexion

Principes de contrôle de congestion

Contrôle de congestion de TCP

UDP



UDP

User Datagram Protocol [RFC 768]

- protocole de transport Internet basique (sans fioriture)
- service *best effort* :
  - ✓ les datagrammes transférés peuvent être...
    - ☞ perdus
    - ☞ dupliqués
    - ☞ désordonnés
- service sans connexion :
  - ✓ pas d'échange préalable
  - ✓ pas d'information d'état aux extrémités
    - ☞ chaque datagramme géré indépendamment

Plan

Rappels sur la couche transport

Multiplexage et démultiplexage

UDP : un protocole en mode non connecté
- **format du datagramme UDP**
- utilisation d'UDP

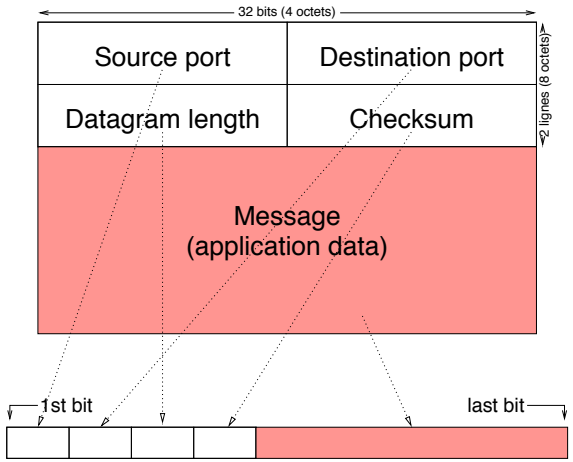
Principes de transfert de données fiable

TCP : un protocole en mode orienté connexion

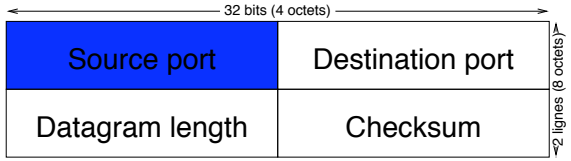
Principes de contrôle de congestion

Contrôle de congestion de TCP

Datagramme UDP

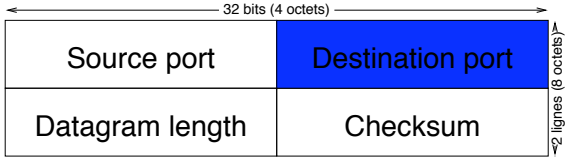


UDP : port source



- 16 bits (65535 ports)
- **multiplexage** à la source
- identification du socket pour un retour potentiel
- allocation fixe ou dynamique (généralement dans le cas d'un client)
- répartition de l'espace des ports :
  - ✓ $0 \leq \texttt{numPort} \leq 1023$: accessible à l'administrateur
    - ☞ socket serveurs (généralement)
  - ✓ $1024 \leq \texttt{numPort}$: accessible aux utilisateurs
    - ☞ socket clients (généralement)

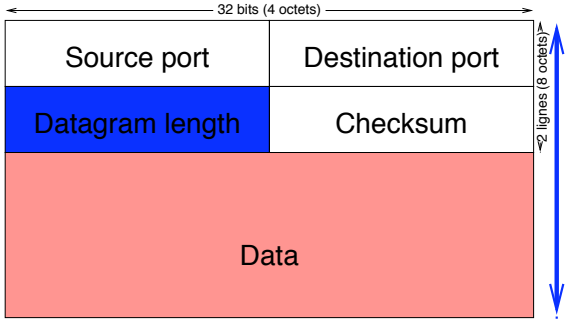
UDP : port destination



- 16 bits (65535 ports)
- **démultiplexage** à la destination
- le destinataire doit être à l'écoute sur ce port
- négociation du port ou *well-known ports* (numéros de port réservés) :

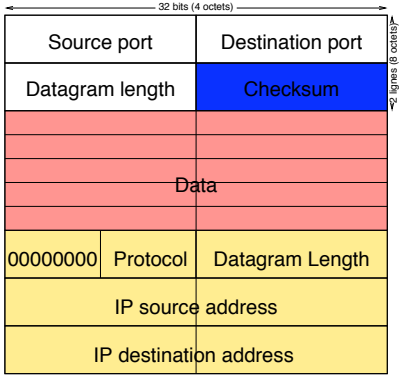
```
Unix> cat \etc\services |grep udp
echo            7/udp
discard         9/udp
daytime         13/udp
chargen         19/udp
ssh             22/udp
time            37/udp
..
..
domain          53/udp
tftp            69/udp
gopher          70/udp
www             80/udp
kerberos        88/udp
snmp            161/udp
snmp-trap       162/udp
....
```


UDP : longueur du datagramme



- 16 bits (64 Koctets maximum)
- longueur totale avec les données exprimée en **octets**

UDP : contrôle d'érreur



- 16 bits
- contrôle d'erreur **facultatif**
- émetteur :
  - ✓ ajout *pseudo-header*
  - ✓ datagram+ = suite $mot_{16\text{bits}}$
  - ✓ *checksum*[2] = $\overline{\sum mot_{16\text{bits}}}$
- récepteur :
  - ✓ ajout *pseudo-header*
  - ✓ recalcul de $\sum mot_{16\text{bits}}$
    - ☞ $= 0$: pas d'erreur détectée toujours possible...
    - ☞ $\neq 0$: erreur (destruction silencieuse)

[2] Somme binaire sur 16 bits avec report de la retenue débordante ajoutée au bit de poid faible

Plan

UDP : arguments pour un transport sans connexion

Le choix d'un service transport non connecté peut être motivé par :
- ressources limitées aux extrémités
  - ✓ pile TCP/IP limitée
  - ✓ absence d'**état** dans les hôtes
  - ✓ capacité de traitement limitée
- besoin d'échange rapide
  - ✓ pas d'**établissement** de connexion
- besoin d'éfficacité
  - ✓ **entête réduit**
- contraintes temporelles
  - ✓ **retransmission** inadapté
  - ✓ pas de **contrôle** du débit d'émission
- besoin de nouvelles fonctionnalités
  - ✓ remontés dans la couche application...

UDP : exemples d'applications

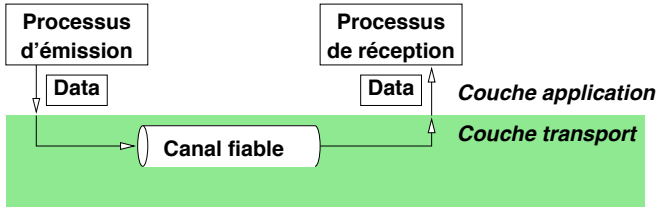
- les applications suivantes reposent typiquement sur UDP :
  - ✓ résolution de noms (DNS)
  - ✓ administration du réseau (SNMP)
  - ✓ protocole de routage (RIP)
  - ✓ protocole de synchronisation d'horloge (NTP)
  - ✓ serveur de fichiers distants (NFS)
    - ☞ fiabilisation implicite par redondance temporelle
    - ☞ fiabilisation explicte par des mécanismes ajoutés dans la couche application
- toutes les applications *multicast* ➠ U.E. **ING**
- et les applications multimédia ➠ U.E. **MMQOS**
  - ✓ diffusion multimédia, *streaming* audio ou vidéo
  - ✓ téléphonie sur Internet
  - ✓ visioconférence
    - ☞ contraintes temporelles
    - ☞ tolérance aux pertes

UDP : Interface socket

```
#include <sys/types.h>
#include <sys/socket.h>

# Create a descriptor
int socket(int domain, int type, int protocol);
#   domain : PF_INET for IPv4 Internet Protocols
#     type : SOCK_DGRAM Supports datagrams (connectionless, unreliable msg of a fixed max length)
# protocol : UDP (/etc/protocols)

# Bind local IP and port
int bind(int s, struct sockaddr *my_addr, socklen_t addrlen);

# Send an outgoing datagram to a destination address
int sendto(int s, const void *msg, size_t len, int flags,
                                    const struct sockaddr *to, socklen_t tolen);

# Receive the next incoming datagram and record is source address
int recvfrom(int s, void *buf, size_t len, int flags,
                                    struct sockaddr *from, socklen_t *fromlen);

# End : dealocate
int close(int s);
```


Plan

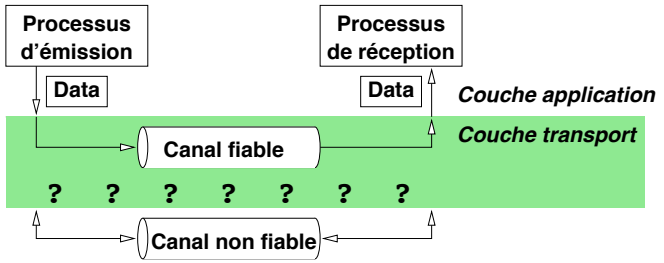
Couche transport et fiablité (1)



Problématique multi-couche :

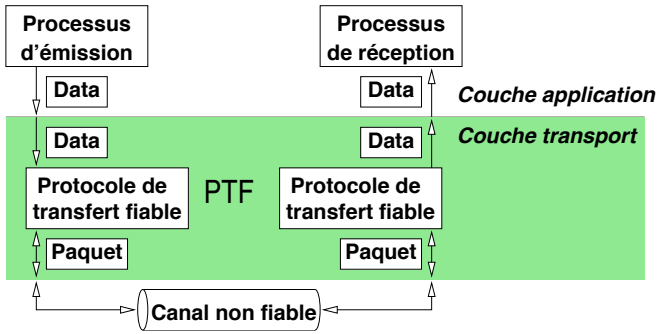
- couche application
- couche transport
- couche liaison

Couche transport et fiablité (2)

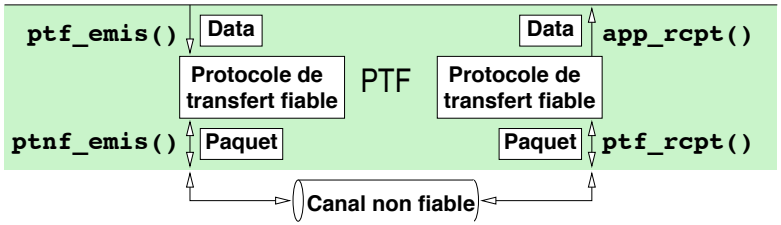


Les cartactéristiques du **canal non fiable** determine la complexité du **protocol de transfert fiable** (PTF).

Couche transport et fiablité (3)



Protocole de Transfert Fiable (PTF)



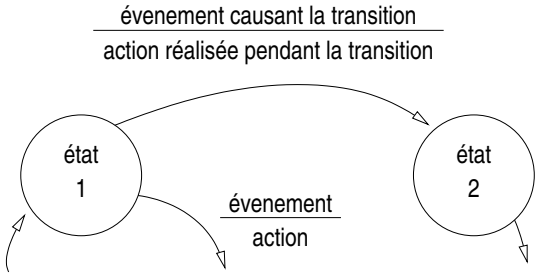
Primitives de base du PTF :

- `ptf_emis()` : appelée par la couche supérieure (application) pour envoyer des données à la couche correspondante du récepteur
- `ptfn_emis()` : appelée par le PTF transférer un paquet sur le canal non fiable vers le récepteur
- `ptf_rcpt()` : appelée lorqu'un paquet arrive au récepteur
- `app_rcpt()` : appelée par le PTF pour livrer les données

PTF et AEF

Nous allons construire progressivement le **PTF**

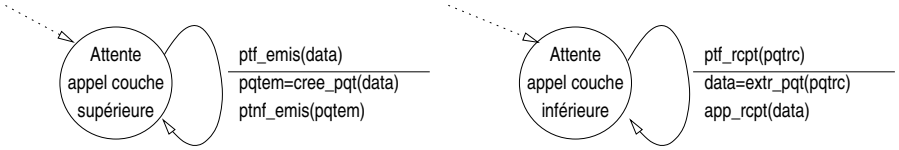
- transfert de données dans un seul sens
  - ✓ information de contrôle dans les 2 directions
- spécification de l'émetteur et du récepteur par des Automates à Etats Finis (AEF) :



PTF v1.0

Transfert fiable sur un canal sans erreur

- canal sous-jacent complètement fiable
  - ✓ pas de bits en erreur
  - ✓ pas de perte de paquets
- automates séparés pour l'émetteur et le récepteur :



émetteur *récepteur*

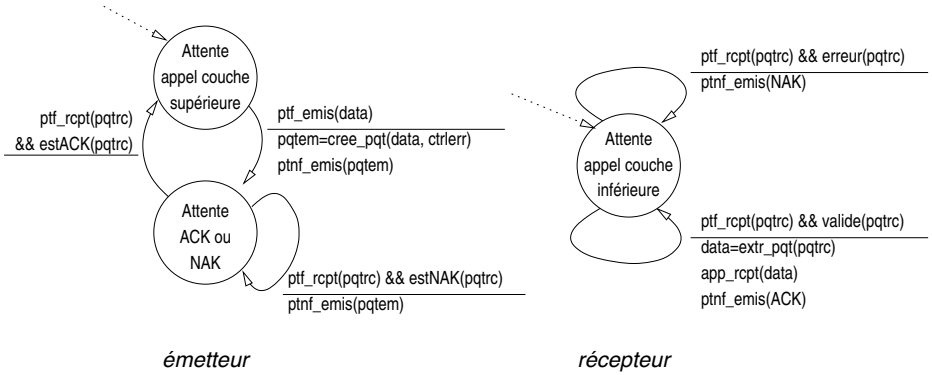
PTF v2.0

Transfert fiable sur un **canal avec des erreurs**

- canal sous-jacent pouvant changer la valeur des bits d'un paquet
  - ✓ introduction de contrôle d'erreur :
    - ☞ `ctrlerr` : redondance rajoutée au paquet
- *Comment récupérer les erreurs ?*
  - ✓ **acquittement** (ACK) : le récepteur indique explicitement la réception correcte d'un paquet
  - ✓ **acquittement négatif** (NAK) : le récepteur indique explicitement la réception incorrecte d'un paquet
    - ☞ l'émetteur ré-émet le paquet sur réception d'un NAK
- nouveau mécanisme dans PTV v2.0 :
  - ✓ détection d'erreur
    - ☞ `valide(pqt)` : vrai si le contrôle d'erreur de `pqt` est correct
    - ☞ `erreur(pqt)` : vrai si le contrôle d'erreur de `pqt` est incorrect
  - ✓ retour d'information *(feedback)* du récepteur (ACK et NAK)

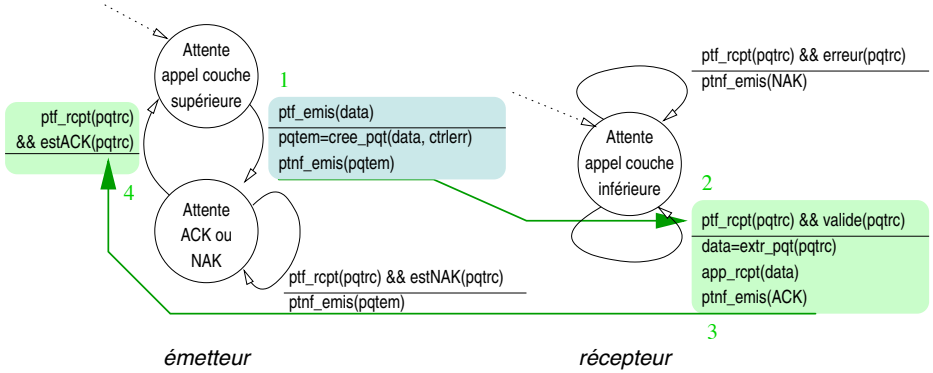
PTF v2.0

Transfert fiable sur un **canal avec des erreurs** :



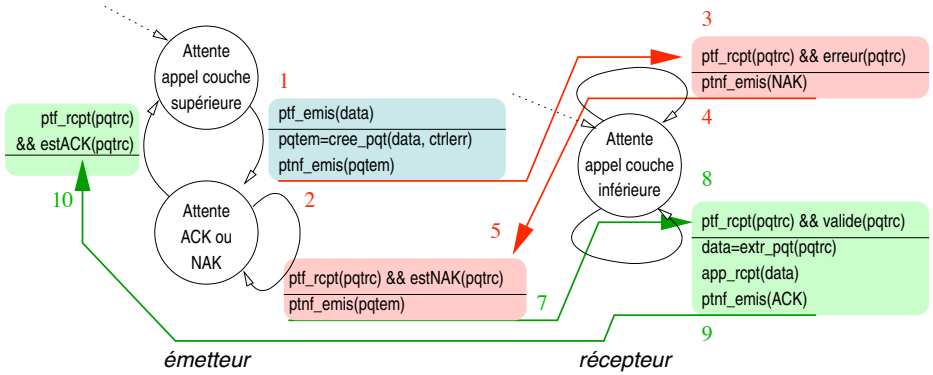
PTF v2.0 : ACK

Transfert fiable lorsqu'il n'y a pas d'erreur :



PTF v2.0 : NAK

Transfert fiable lorqu'il y a une erreur :



PTF v2.0 : discussion

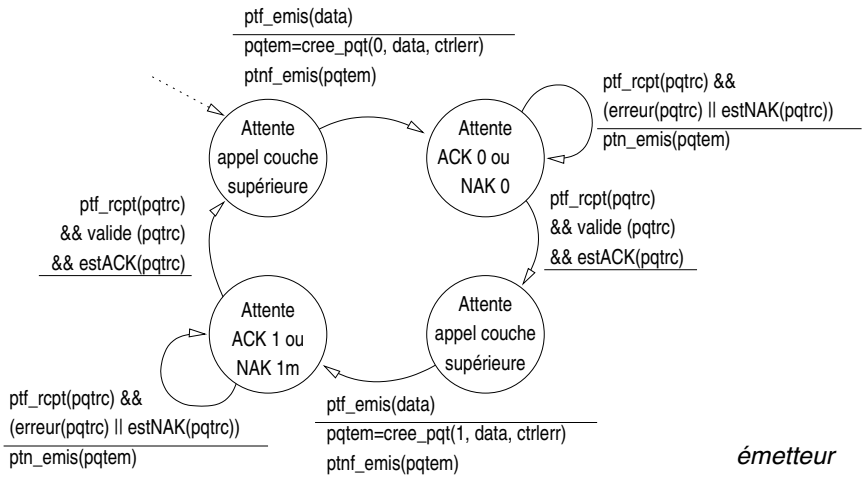
PTF v2.0 est un protocole *stop and wait* :

- émetteur envoi un paquet et attend la réponse du récepteur
- peu performant...

PTF v2.0 à un défaut majeur !

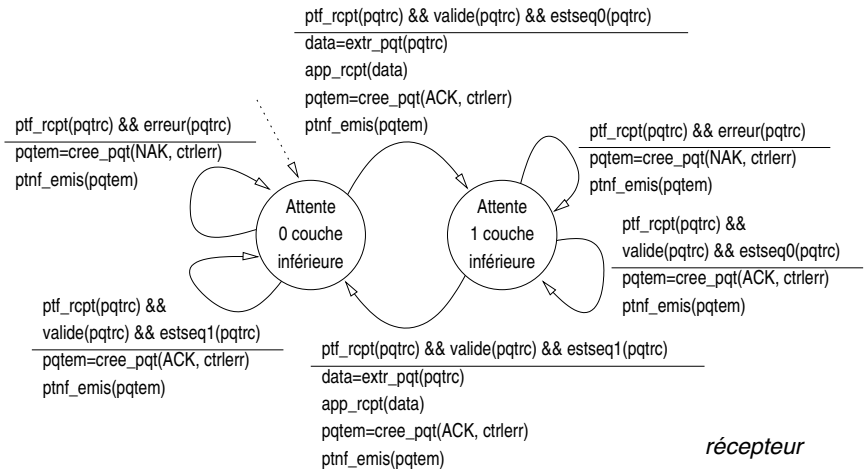
- *Que se passe-t-il si les ACK ou NAK sont incorrect ?*
  - ✓ pas d'information sur l'état du récepteur
  - ✓ une retransmission simple risque de dupliquer les données
- gestion des duplicats :
  - ✓ émetteur **retransmet** le paquet courant si ACK/NAK incorrect
  - ✓ émetteur insert un **numéro** de séquence à chaque paquet
  - ✓ récepteur **supprime** les paquets dupliqués
    - ☞ inclu dans **PTF v2.1**

PTF v2.1 : émetteur



émetteur

PTF v2.1 : récepteur



récepteur

PTF v2.1 : discussion

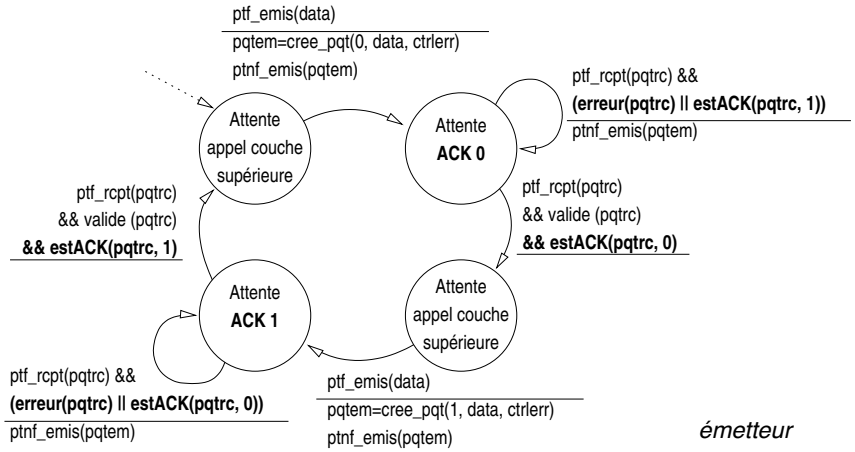
Comportement des extrémités avec PFT v2.1

- **émetteur**
  - ✓ ajout de numéro de séquence à chaque paquet
    - ☞ 2 suffisent (0 et 1)
  - ✓ contrôle d'erreur sur les ACK et NAK
  - ✓ 2 fois plus d'états
- **récepteur**
  - ✓ vérification que le paquet n'est pas dupliqué
    - ☞ l'état où l'on se trouve indique le numéro de séquence attendu

Peut-on éliminer les NAK ?

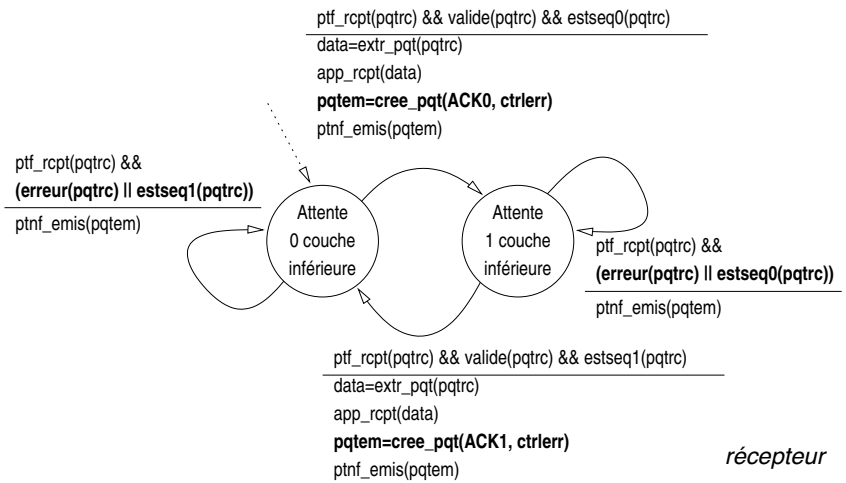
- remplacement des NAK par **ACK du dernier paquet** valide recu
  - ✓ récepteur inclu le numéro de séquence correspondant dans le ACK
  - ✓ ACK dupliqué au récepteur = NAK recu au récepteur
    - ☞ intégré dans **PFT v2.2**

PTF v2.2 : émetteur



émetteur

PTF v2.2 : récepteur



récepteur

PTF v3.0

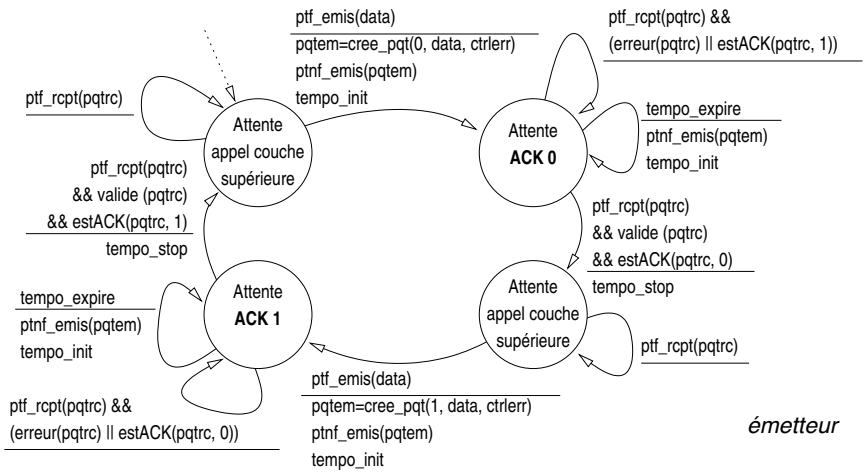
Transfert fiable sur un **canal avec erreurs et pertes**

- canal sous-jacent peut aussi perdre des paquets (data ou ACK)
  - ✓ `ctrlerr` + `numSeq` + ACK + retransmission
    - ☞ insuffisant : l'absence d'un paquet bloque l'automate !

Temporisation des retransmission

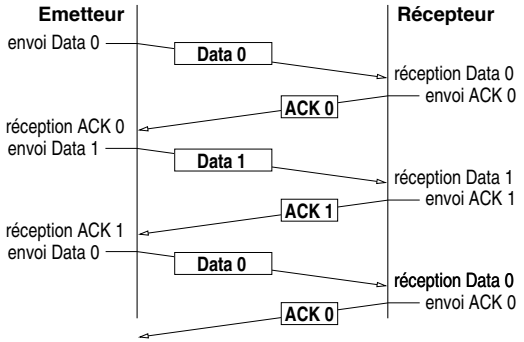
- estimation d'un temps de retour de ACK raisonable
  - ✓ déclenchement d'une temporisation à l'emission d'un paquet
    - ☞ `tempo_init`
  - ✓ ACK avant l'expiration de la temporisation ➠ rien
    - ☞ `tempo_stop`
  - ✓ pas de ACK à l'expiration de la temporisation ➠ retransmission
    - ☞ `tempo_expire`
- si le ACK est seulement en retard...
  - ✓ retransmission = duplication
    - ☞ détectée grâce au numéro de séquence

PTF v3.0 : émetteur

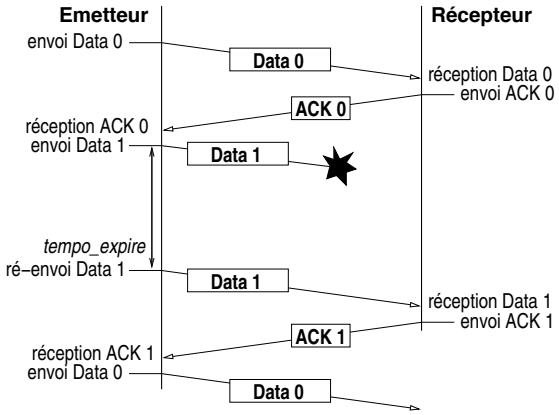


émetteur

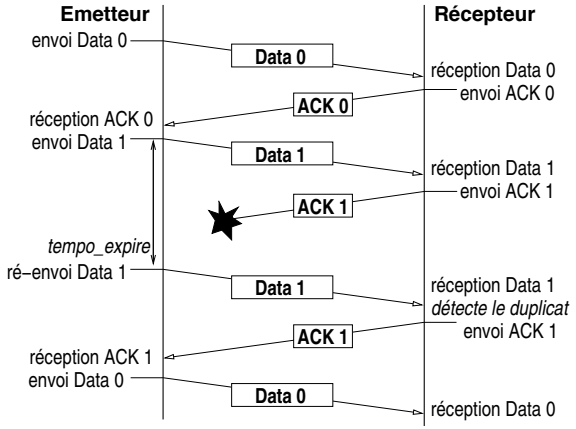
PTF v3.0 : sans perte



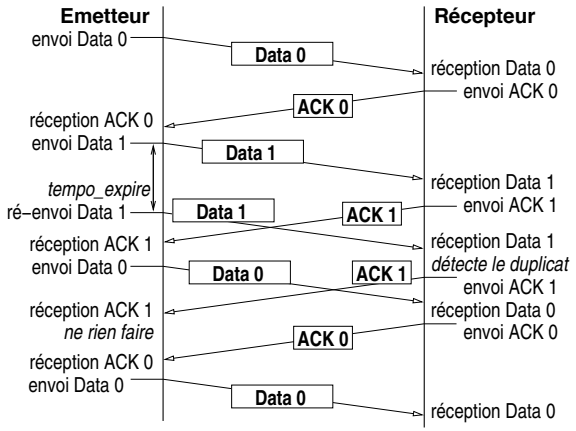
PTF v3.0 : perte d'un paquet de données



PTF v3.0 : perte d'un ACK



PTF v3.0 : fin de temporisation prématurée

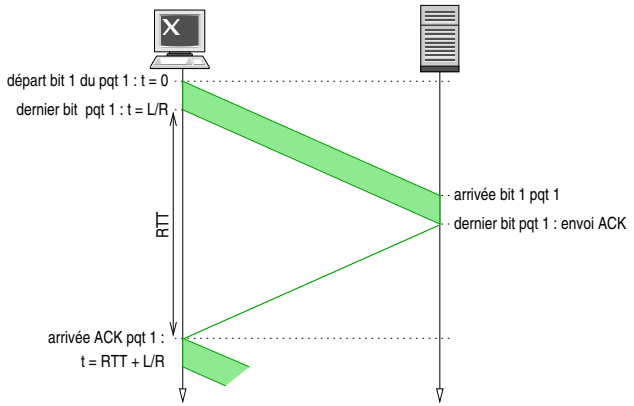


PTF v3.0 : performance

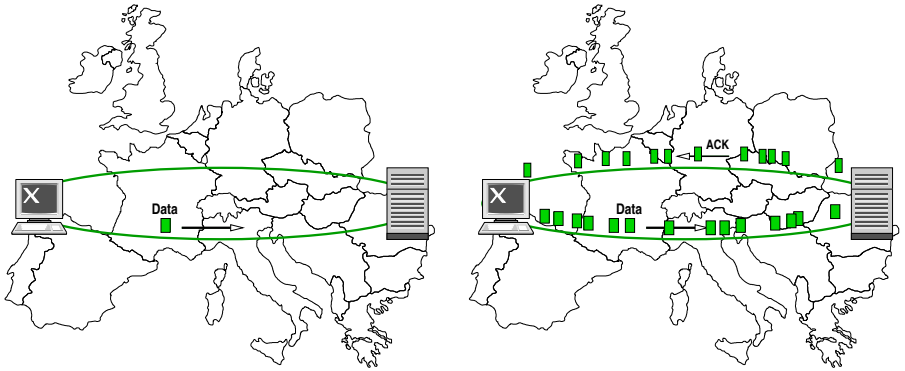
PFT v3.0 fonctionne mais quelles sont ses performances ?

- exemple de communication :
  - ✓ débit du lien : $D_{reseau} = 1\ Gbps$,
  - ✓ délais de bout-en-bout : $d = 40\ ms$ ($d_{AR} = 80\ ms$)
  - ✓ paquets de longueur 1000 octets ($L_{paquet} = 8000\ b$)
- $T_{transmission} = L_{paquet}/D_{reseau} = 8.10^3/10^9 = 8\ \mu s$
- efficatité émetteur (E_{emis}) : fraction de temps en émission
  - ✓ $E_{emis} = \frac{L_{paquet}/D_{reseau}}{L_{paquet}/D_{reseau}+d_{AR}} = \frac{8.10^{-6}}{8.10^{-6}+8.10^{-2}} = \frac{1}{10000}$
  - ✓ $D_{transport} = L_{paquet}/d_{AR} = 8.10^3/8.10^{-2} = 100\ Kbps$
    - ☞ **le protocole limite l'utilisation des ressources disponibles**

PTF v3.0 : stop and wait

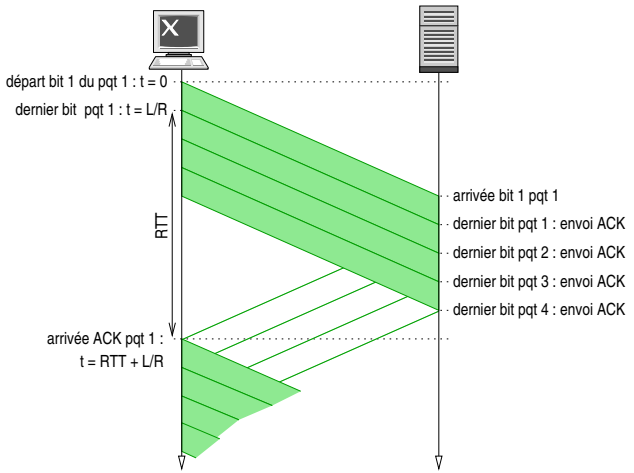


Protocole pipeline



- l'émetteur autorise plusieurs paquets en attente d'acquittement
  - ✓ numéro de sequences étendus
  - ✓ tampons d'émission et de réception
    - ☞ 2 types de protocole pipeliné : **Go-Back-N** et **Retransmissions sélectives**

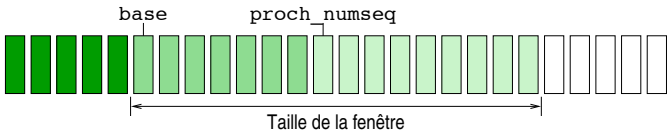
Performance pipeline



Go-Back-N : émetteur

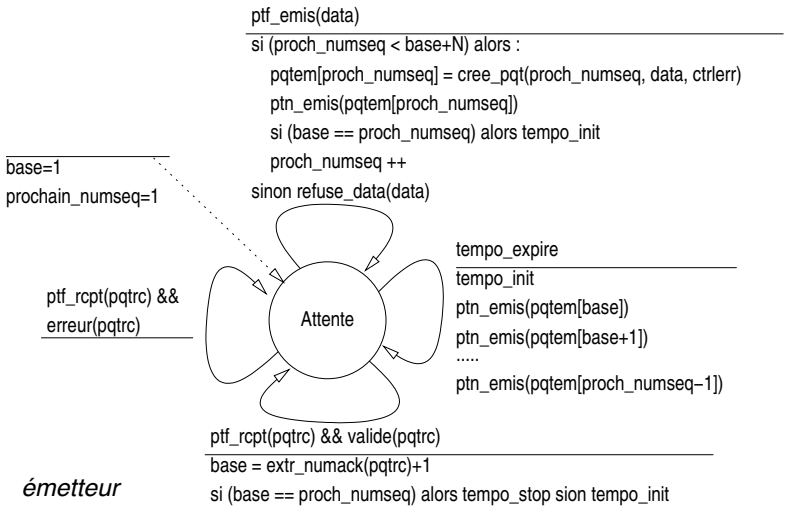
Emetteur avec gestion *Go-Back-N* (retour arrière).

- entête des paquets avec k bits de numéro de séquence
- acquittements **cumulatifs**
  - ✓ `ACK(n)` acquitte tous les paquets jusqu'au numéro de séquence `n`
- fenêtre d'au maximun *N* paquets non acquités :



- temporisation pour chaque paquet en attente *(in-flight)*
  - ✓ `tempo_expire(n)` : retransmission du paquet `n` et des suivants avec numéro de séquence supérieur

PTF v4.0 : émetteur



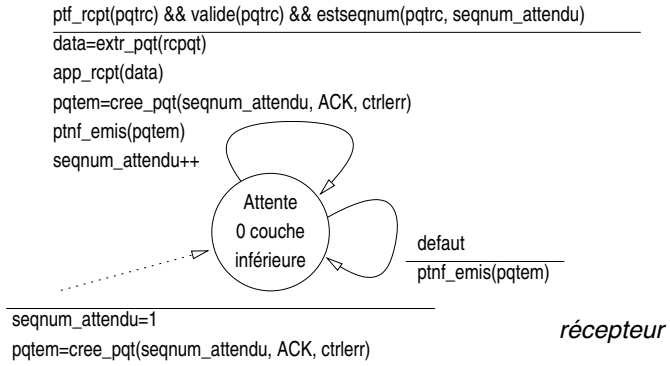
émetteur

Go-Back-N : Récepteur

Récepteur avec gestion *Go-Back-N* (retour arrière).

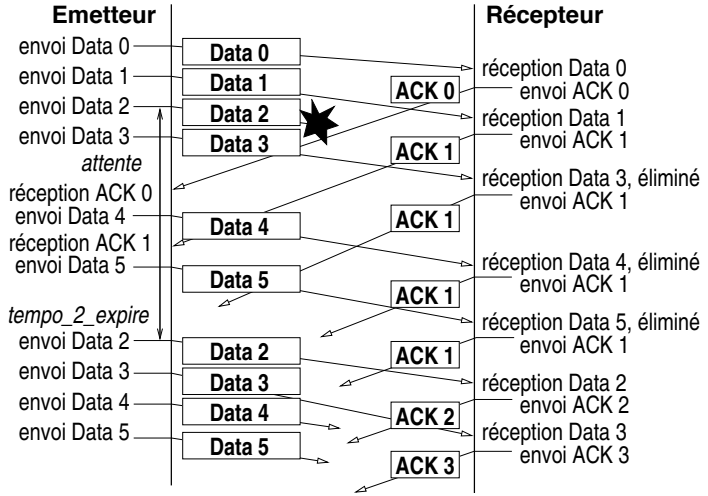
- **seulement des ACK** :
  - ✓ envoie toujours des ACK avec le plus élevé des `seqnum` de paquets valides **ordonnés**
    - ☞ peut générer des ACK dupliqués
    - ☞ seul `seqnum_attendu` est mémorisé
- **déséquencement** :
  - ✓ élimine les paquets déséquencés
    - ☞ pas de tampon au niveau du récepteur
  - ✓ ré-émet le ACK avec le plus élevé des `seqnum` de paquets valides **ordonnés**

PTF v4.0 : récepteur



récepteur

PTF v4.0 : exemple



Retransmissions sélectives (1)

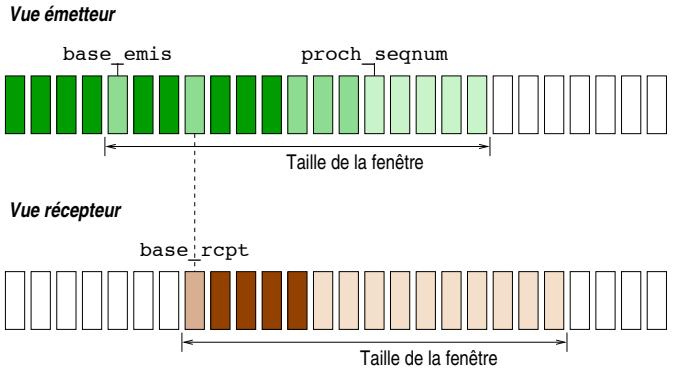
Emetteur

- retransmet **seulement** les paquets non acquittés
- fenêtre d'émission limité à *N* paquets consécutifs
- algo :
  - ✓ `pft_emis(data)`
    - ☞ envoi un paquet si `seqnum` dans la fenêtre
  - ✓ `tempo_expire(n)`
    - ☞ retransmet paquet `n`
    - ☞ `tempo_init(n)`
  - ✓ `ACK(n)`
    - ☞ marque le paquet `n` reçu
    - ☞ si `n` est le plus petit paquet non acquitté, décale la fenêtre

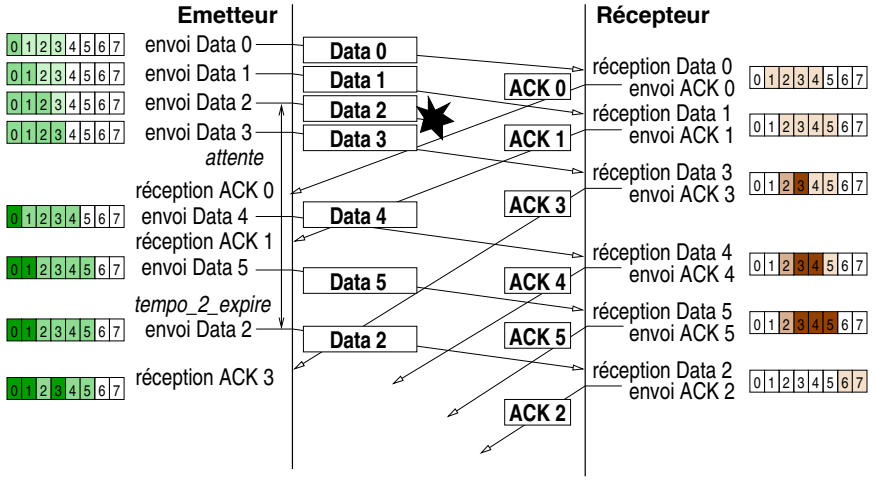
Récepteur

- acquitte **explicitement** chaque paquet valide reçu
- tampon de réception pour reséquencement
- algo :
  - ✓ `ptf_rcpt(n)`
    (`base_rcpt` $\leq$ `n` $\leq$ `base_rcpt`+N-1)
    - ☞ `ACK(n)`
    - ☞ si déséquensé : tampon
    - ☞ si séquense : `app_emis(data)`, est le plus petit paquet non acquitté, décale la fenêtre
  - ✓ `ptf_rcpt(n)`
    (`base_rcpt-N` $\leq$ `n` $\leq$ `base_rcpt-1`)
    - ☞ `ACK(n)`
  - ✓ autre
    - ☞ ignore

Retransmissions sélectives (2)



Retransmissions sélectives (3)



Plan

Rappels sur la couche transport

Multiplexage et démultiplexage

UDP : un protocole en mode non connecté

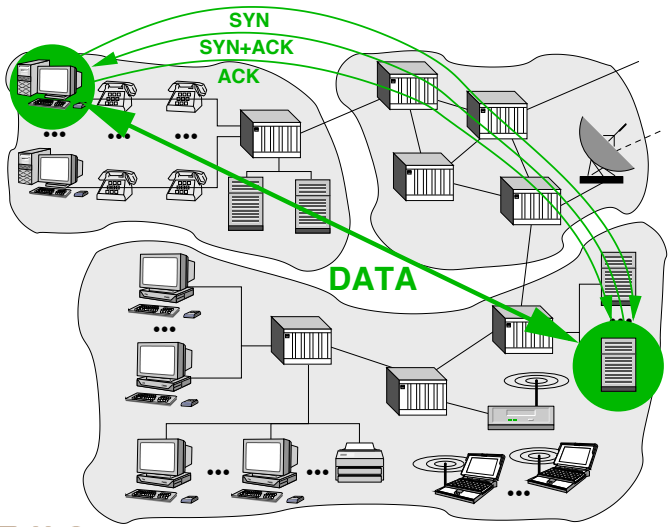
Principes de transfert de données fiable

TCP : un protocole en mode orienté connexion
- format du segment TCP
- gestion de la connexion
- calcul des temporisations
- mise en œuvre de la fiabilité
- contrôles de flux
- utilisation de TCP

Principes de contrôle de congestion

Contrôle de congestion de TCP

TCP (1)



TCP (2)

Transmission Control Protocol [RFCs : 793, 1122, 1323, 2018, 2474, 2581, 3168 et 4379]

- service **fiable**
  - ✓ mécanismes ARQ
- **point-à-point**
  - ✓ deux processus (généralement un client et un serveur)
- flot d'**octet** continu
  - ✓ pas de frontières de messages
- **orienté connexion**
  - ✓ ouverture en trois échanges (*three-way handshake*)
    - ☞ initiation des états aux extrémité avant l'échanges de données
  - ✓ fermetures courtoise ou brutale
- connexion **bidirectionnelle** (*full duplex*)
  - ✓ flux de données dans chaques sens
  - ✓ taille maximum du segment : MSS (*Maximun Segment Size*)
- **pipeline**
  - ✓ tampons d'émission et de réception
  - ✓ double fenêtre asservie aux contrôles de flux et de congestion

Plan

Rappels sur la couche transport

Multiplexage et démultiplexage

UDP : un protocole en mode non connecté

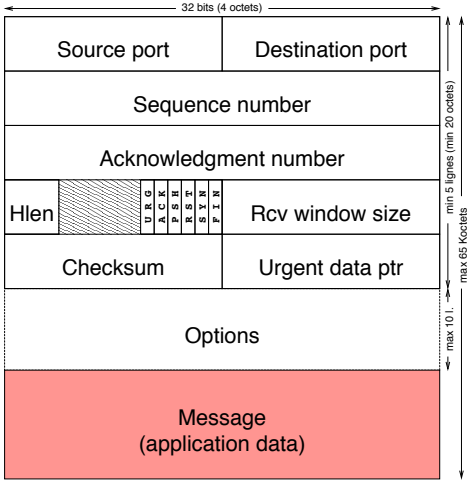
Principes de transfert de données fiable

TCP : un protocole en mode orienté connexion
- **format du segment TCP**
- gestion de la connexion
- calcul des temporisations
- mise en œuvre de la fiabilité
- contrôles de flux
- utilisation de TCP

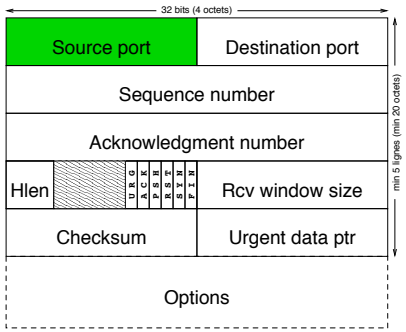
Principes de contrôle de congestion

Contrôle de congestion de TCP

Segment TCP

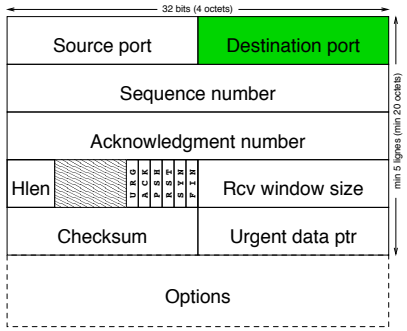


TCP : Port source



- 16 bits (65535 ports)
- **multiplexage** à la source
- identification partielle du socket (demi-association locale)
- allocation fixe ou dynamique (généralement dans le cas d'un client)
- répartition espace des ports :
  - ✓ $0 \leq \texttt{numPort} \leq 1023$: accessible à l'administrateur
    - ☞ socket serveurs (généralement)
  - ✓ $1024 \leq \texttt{numPort}$: accessible aux utilisateurs
    - ☞ socket clients (généralement)

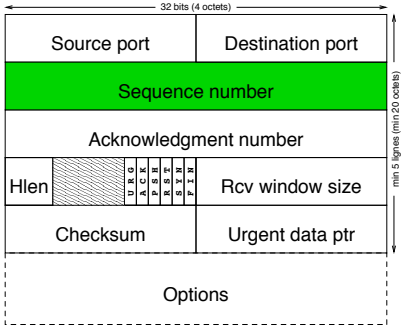
TCP : Port destination



- 16 bits (65535 ports)
- **démultiplexage** au niveau de la destination
- identification partielle du socket (demi-association distante)
- lors de la cration de l'association, le destinataire doit être à l'écoute sur ce port
- négociation du port ou *well-known ports* (numéros de port réservés) :

```
Unix> cat \etc\services|grep tcp
tcpmux          1/tcp
discard         9/tcp
systat          11/tcp
chargen         19/tcp
ftp-data        20/tcp
ftp             21/tcp
ssh             22/tcp ..
telnet          23/tcp
smtp            25/tcp
whois           43/tcp
domain          53/tcp
gopher          70/tcp
finger          79/tcp
www             80/tcp
kerberos        88/tcp ....
```

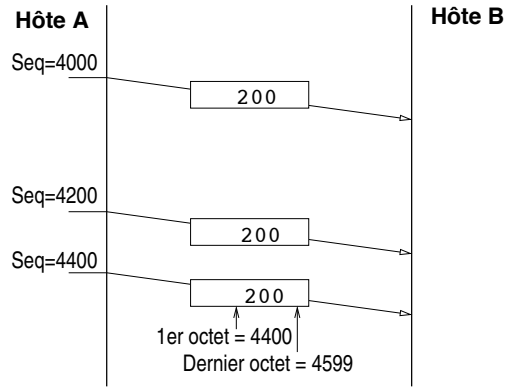

TCP : Numéro de séquence (1)



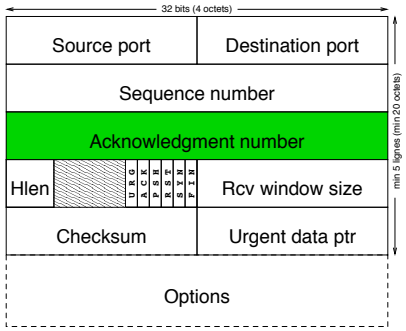
- 32 bits
- associé à chaque **octet** (et non pas à un segment)
  - ✓ numérote le **premier** octet des *data*
  - ✓ numérotation implicite des octets suivants
  - ✓ boucle au bout de 4 Goctets
- détection des **pertes**
- **ordonnancement**

TCP : Numéro de séquence (2)

Numérotation de chaque **octet** du flot continu de données



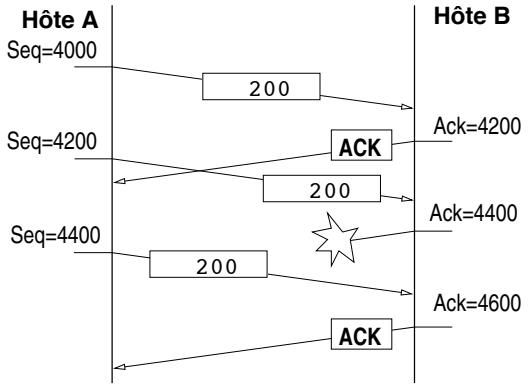
TCP : Numéro d'acquittement (1)



- 32 bits
- *piggybacking*
- indique le numéro du **prochain** octet attendu
- **cumulatif**, indique le premier octet non reçu (d'autres peuvent avoir été reçus avec des numéros de séquence supérieurs)

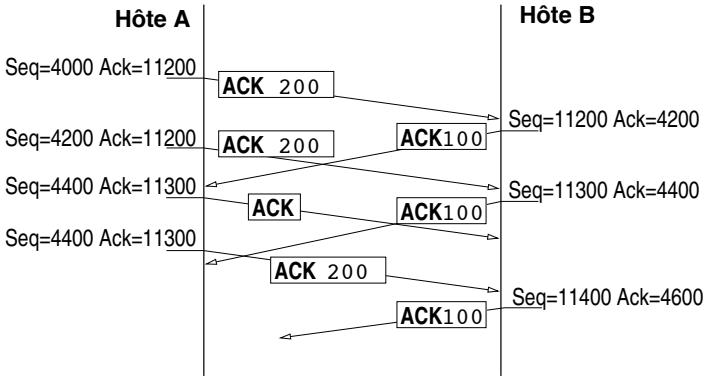
TCP : Numéro d'acquittement (2)

Acquittement de chaque **octet** du flot continu de données



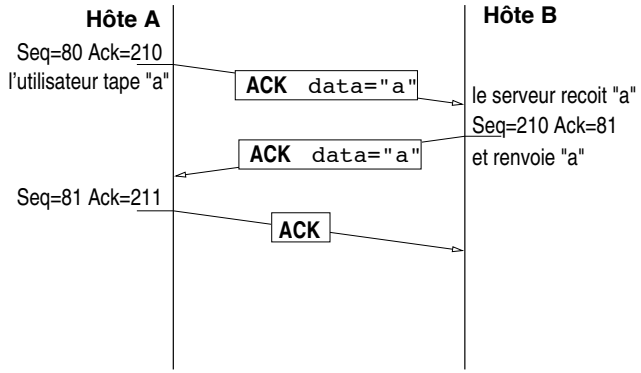
TCP : Numéro d'acquitement (3)

Piggybacking



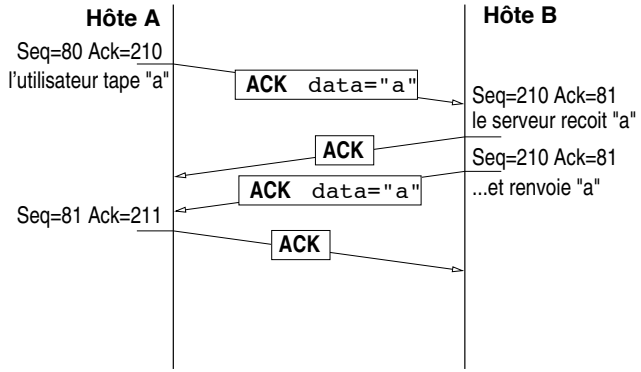
TCP : Exemple TELNET (1)

Emission d'un caractère frappé et renvoi par le serveur pour l'affichage



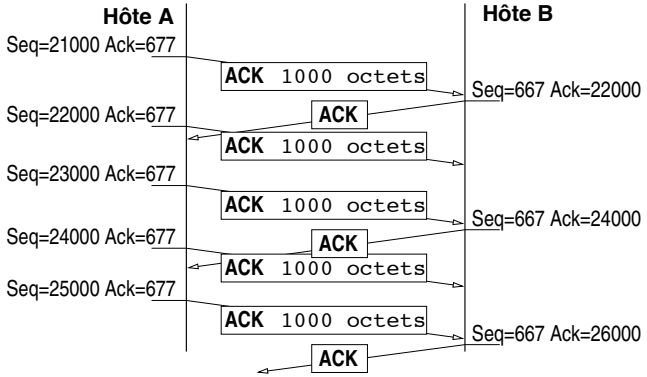
TCP : Exemple TELNET (2)

Les acquittements peuvent être plus rapide que l'application

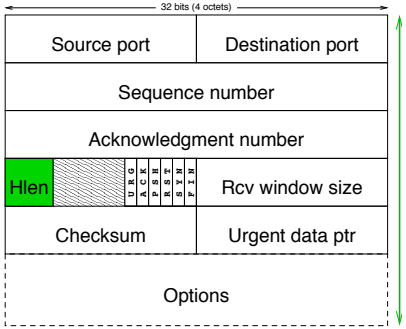


TCP : Acquittements temporisés

Delayed ACK (attente de deux segments ou 500 ms max.)

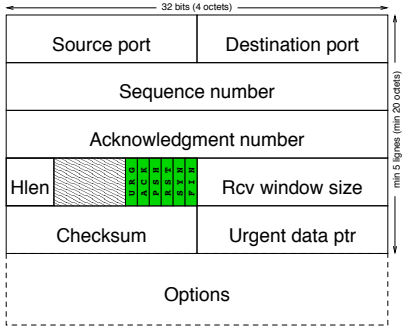


TCP : Longueur de l'entête



- 4 bits (valeur 15 max)
- nombre de lignes de 32 bits dans l'entête TCP
- nécessaire car le champ option est de longueur variable
  - ✓ valeur 5...
    - ☞ pas d'options
    - ☞ entête TCP de 20 octets minimum
  - ✓ ... à 15
    - ☞ 10 lignes d'options
    - ☞ 40 octets d'options max
    - ☞ entête TCP de 60 octets maximum

TCP : Indicateurs (*flags*)

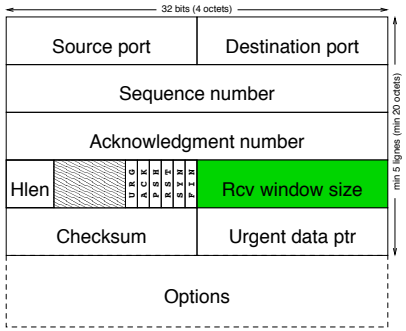


Chacun sur 1 bit indique :

- `URG` : présence de données **urgentes**
- `ACK` : le champ **acquittement** est valide
- `PSH` : envoi **immédiat** avec vidage des tampons
- `RST` : **terminaison** brutale de la connexion
- `SYN` : synchronisation lors de l'**ouverture**
- `FIN` : echanges terminaux lors d'une **fermeture** courtoise

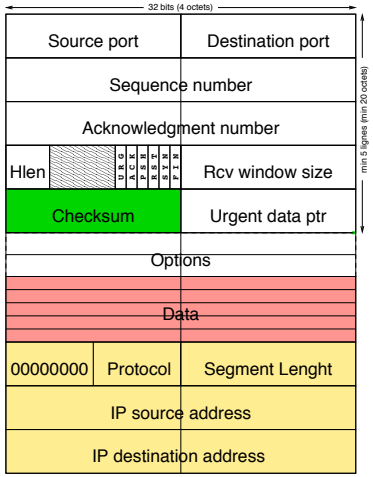
✓ il y en a d'autres récents
➠ U.E. **ING**

TCP : Taille de la fenêtre de réception



- 16 bits
  - ✓ le récepteur peut annoncer jusqu'à 64 Koctets
- *piggybacking*
- **contrôle de flux**
  - ✓ indique le nombre d'octets disponibles du coté du récepteur
  - ✓ dimentionne la taille de la fenêtre d'anticipation de l'émetteur

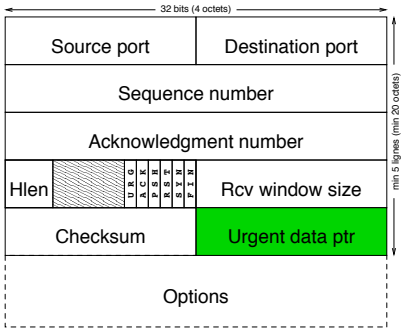
TCP : Somme de contrôle du segment



- 16 bits
- contrôle d'erreur (idem UDP)
- émetteur :
  - ✓ ajout *pseudo-header*
  - ✓ datagram$+=$ suite $mot_{16\text{bits}}$
  - ✓ *checksum*[3] $= \overline{\sum mot_{16\text{bits}}}$
- récepteur :
  - ✓ ajout *pseudo-header*
  - ✓ recalcul de $\sum mot_{16\text{bits}}$
    - ☞ $= 0$: pas d'erreur détectée toujours possible...
    - ☞ $\neq 0$: erreur (destruction silencieuse)

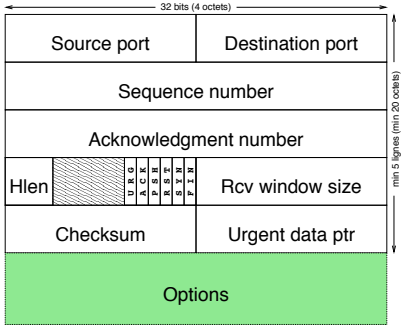
[3] Somme binaire sur 16 bits avec report de la retenue débordante ajoutée au bit de poid faible

TCP : Pointeur sur les données urgentes



- 16 bits
- permet l'envoi de données spéciales (et non **hors bande**)
- délimite des données traitées en priorité
- indique la fin des données urgentes
  - ✓ interprétation de la quantité de données et de leur rôle par l'application

TCP : Options



Les options sont de la forme TLV ou *Type, Length* (octets), *Value* :

- `END` : fin de liste d'options (T=0, non obligatoire)
- `NOOP` : pas d'opération (T=1, bourrage)
- `MSS` : négociation du MSS (T=2, L=4, V=MSS)
- `WSIZE` : mise à l'échelle de la fenêtre par le facteur 2^N (T=3, L=3, V=N)
- `SACK` : demande d'acquittement sélectif (T=4,L=2, à l'ouverture)
- `SACK` : acquittement sélectif de n blocs (T=5, L=$2+8n$, $2n$ numéros de séquences)
- ...

Plan

TCP : Gestion de la connexion

Ourverture (création) de la **connexion** préalable à l'échange des données :

- initialisation des variables TCP
  - ✓ synchronisation des numéros de séquence
  - ✓ création des tampons
  - ✓ initiation du controle de flot
- **client** : initiateur de la connexion
- **serveur** : en attente de la demande de connexion

Fermeture (terminaison) de la connexion après l'échange des données :

- attente ou non de l'émission des données restantes
- libération des tampons

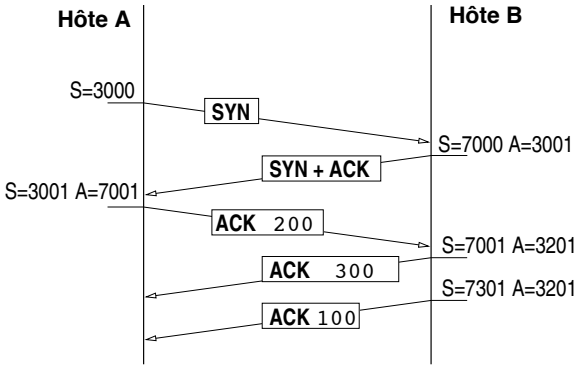
TCP : *Three-Way Handshake* (1)

Echange initial en 3 segments *(Three-Way Handshake)*

- **1** client ➠ serveur : segment TCP avec le bit `SYN`
  - ✓ indique le numéro de séquence initial (ISN) choisi par le client
  - ✓ l'emission du `SYN` incrémentera le futur numéro de séquence
  - ✓ pas de données
- **2** serveur ➠ client : segment TCP avec les bits `SYN` + `ACK`
  - ✓ la réception du `SYN` à incrémenté le numéro de d'aquittement
  - ✓ indique le numéro de séquence initial (ISN) choisi par le serveur
  - ✓ l'emission du `SYN` incrémentera le futur numéro de séquence
  - ✓ allocation des tampons du serveur
- **3** client ➠ serveur : segment TCP avec le bit `ACK`
  - ✓ la réception du `SYN` à incrémenté le numéro de d'aquittement
  - ✓ peut contenir des données

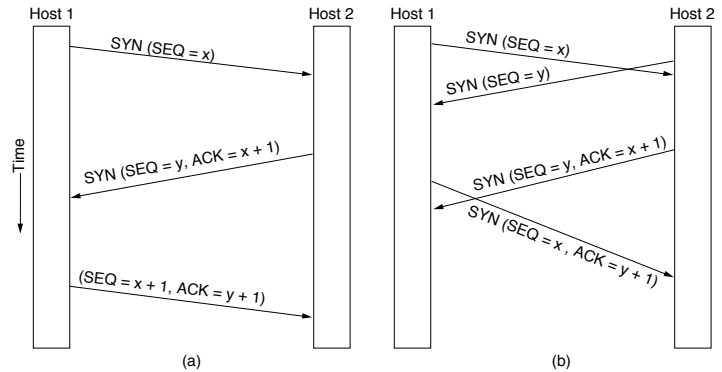
TCP : *Three-Way Handshake* (2)

Echange initial en 3 segments



TCP : *Three-Way Handshake* (3)

Gestion des ouvertures simultanées



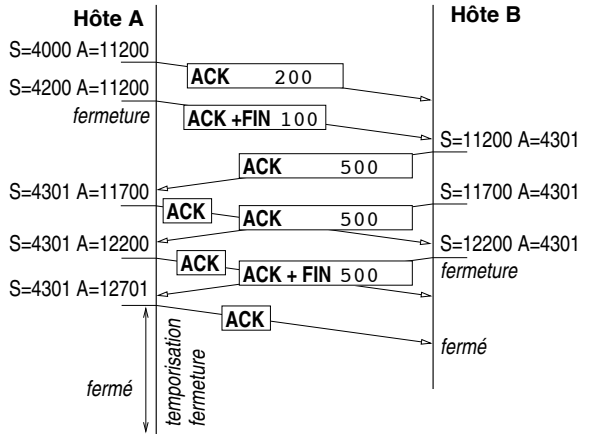
pictures from TANENBAUM A. S. *Computer Networks 3rd edition*

TCP : *Gracefull Release* (1)

- 1 le **client** émet un segment TCP avec `FIN`
  - ✓ l'émission du `FIN` incrémentera le futur numéro de séquence
  - ✓ peut contenir des données
- 2 le **serveur** recoit le segment avec `FIN`
  - ✓ la réception du `FIN` incrémente le numéro d'aquittement
  - ✓ émet un segment TCP avec `ACK`
  - ✓ termine la connexion **(envoie les données restantes)**
  - ✓ émet un segment TCP avec `FIN`
  - ✓ l'émission du `FIN` incrémentera le futur numéro de séquence
- 3 le **client** recoit le segment avec `FIN`
  - ✓ la réception du `FIN` incrémente le numéro d'aquittement
  - ✓ emet un segment TCP avec `ACK`
  - ✓ termine la connexion
    - ☞ déclanche une temporisation d'attente (`FIN` dupliquées)
- 4 le **serveur** recoit le segment avec `FIN`

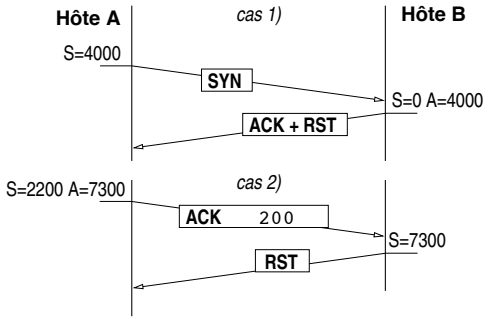
TCP : *Gracefull Release* (2)

Déconnexion : terminaison courtoise

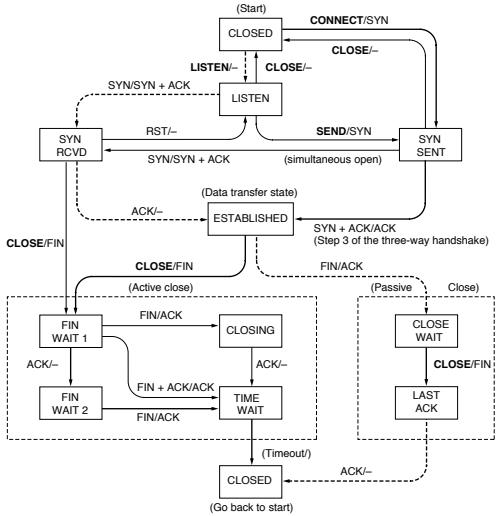


TCP : *Shutdown*

Déconnexion : terminaison unilatérale
(pour tout comportement anormal ou indésiré)



TCP : Automate d'états finis



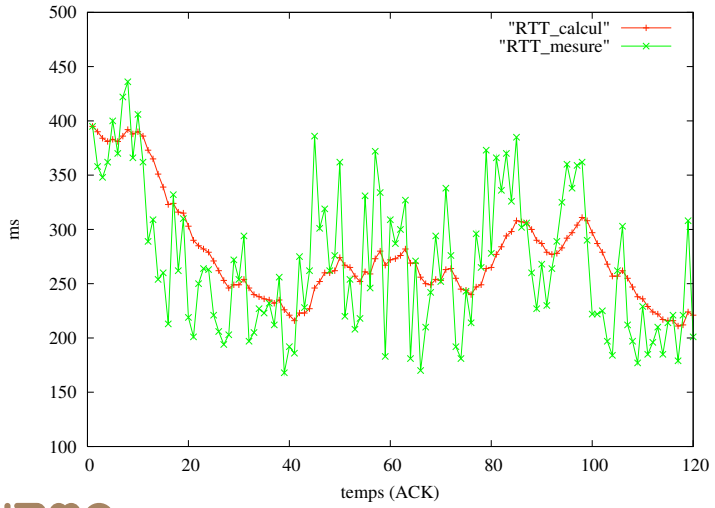
Plan

TCP : Calcul du RTT

Round Trip Time

- Estimation de la temporisation de retransmission :
  - ✓ supérieure au RTT... mais le RTT varie !
    - ☞ trop petit : retransmissions inutiles
    - ☞ trop grand : réaction lente aux pertes
- Estimation du RTT :
  - ✓ $RTT_{mesure} = \Delta$ (envoi segment, reception ACK correspondant)
  - ✓ RTT_{mesure} peut varier rapidement ➠ lissage
    - ☞ $RTT = \alpha RTT_{mesure} + (1 - \alpha) RTT_{ancien}$
      avec α usuel $= 1/8$
  - ✓ moyenne glissante à décroissance exponentielle

TCP : Exemple de calcul de RTT



TCP : Temporisations

Gestion de multiples temporisations (*timers*) :

- *retransmission timer* (détecte les pertes)
  - ✓ $RTO = RTT + \delta D$
    - ☞ avec $\delta = 4$ et une valeur initiale du RTT élevée (3 secondes)
  - ✓ $D = \beta(|RTT_{mesure} - RTT_{ancien}|) + (1 - \beta)D_{ancien}$
    - ☞ calcul de l'écart moyen avec β usuel $= 1/4$
  - ✓ **algorithme de Karn**
    - ☞ ne pas tenir compte des paquets retransmis et doubler le RTO à chaque échec (*exponential backoff*)
- *persistence timer* (évite les blocages)
  - ✓ envoi d'un acquittement avec une fenêtre à 0
- *keep alive timer* (vérifie s'il y a toujours un destinataire)
- *closing timer* (terminaison)

Plan

Transmission fiable de TCP

TCP est un protocole fiable de transfert sur le service IP non fiable

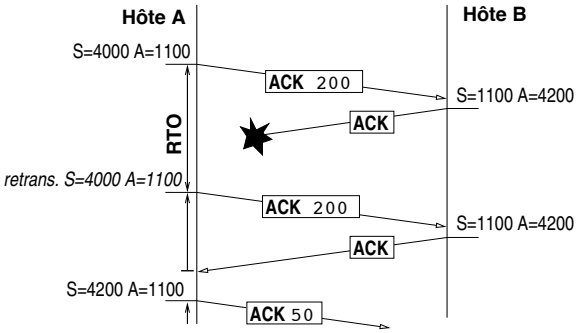
- mécanismes de base :
  - ✓ **pipeline**
  - ✓ ACK **cumulatifs**
  - ✓ temporisateur de retransmission **unique**
  - ✓ retransmissions déclanchées par :
    - ☞ expiration de temporisation *(timeout)*
    - ☞ duplication d'ACK
- dans la suite...
  - ✓ émetteur TCP simplifié :
    - ☞ pas d'ACK dupliqué
    - ☞ pas de contrôle de flux
    - ☞ pas de contrôle de congestion

Évènements émetteur TCP

- **réception des données de la couche supérieure**
  - ✓ **création** d'un segment avec `numSeq`
    - ☞ `numSeq` est le numéro dans le flux d'octet du premier octet de donnée du segment
  - ✓ démarrer la **temporisation** si elle n'est pas déjà en cours
    - ☞ la temporisation correspond au segment non acquitté le plus ancien
- **expiration de temporisation** *(timeout)*
  - ✓ **retransmission** du segment associé à la temporisation
  - ✓ redémarrer la **temporisation**
- **réception d'aquittement** (ACK)
  - ✓ si acquitte des segments non acquités :
    - ☞ actualiser la base de la fenêtre de transmission (`base_emis`)
    - ☞ redémarrer la **temporisation** si d'autres ACK sont attendus

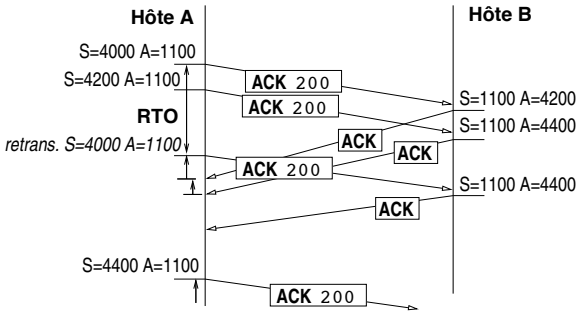
Retransmission TCP (1)

Scénario avec ACK perdu



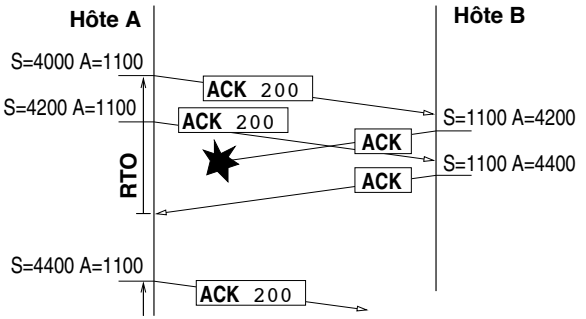
Retransmission TCP (2)

Scénario avec temporisation sous-estimée



Retransmission TCP (3)

Scénario avec ACK cumulatifs



Évènement récepteur TCP

Génération d'**ACKs** (actions du récepteur)

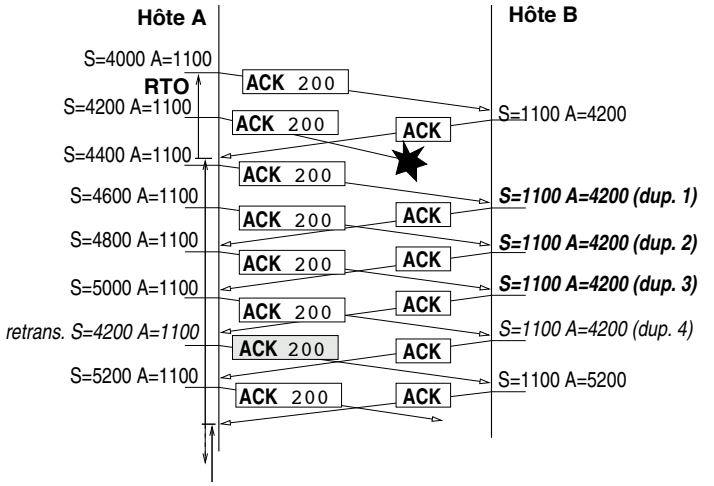
- arrivée d'un segment **dans l'ordre** avec le `numSeq` attendu :
  - ✓ les segments précédents sont déjà acquittés
    - ☞ ACK **retardé** *(delayed ACK)*, attente jusqu'à 500 ms
    - ☞ si pas d'autre segments, envoi d'un ACK
  - ✓ un autre segment est en attente d'acquittement
    - ☞ envoi immédiat d'un ACK **cumulatif** pour ces deux segments dans l'ordre
- arrivée d'un segment **dans le désordre** :
  - ✓ `numSeq` supérieur à celui attendu (intervalle détecté)
    - ☞ envoi immédiat d'un ACK **dupliqué**
    - ☞ rappel du prochain `numSeq` attendu
  - ✓ rempli partiellement ou totalement un intervalle
    - ☞ envoi immédiat d'un ACK
    - ☞ nouveau `numSeq` attendu suite au remplissage de l'intervalle

TCP : *Fast Retransmit* (1)

Optimisation du mécanisme de retransmission

- temporisation souvent relativement élevée
  - ✓ délai important avant une retransmission
- détection des segments perdus grâce aux ACKs **dupliqués**
  - ✓ ensemble de segments souvents envoyés cote-à-cote
  - ✓ si un segment est perdu ➠ nombreux ACKs dupliqués
- si l'émetteur reçoit 3 ACK dupliqués (4 ACKs identiques)
  - ✓ TCP suppose que le segment suivant celui acquité est perdu
    - ☞ **fast retransmit** : retransmission du segment avant l'expiration de la temporisation

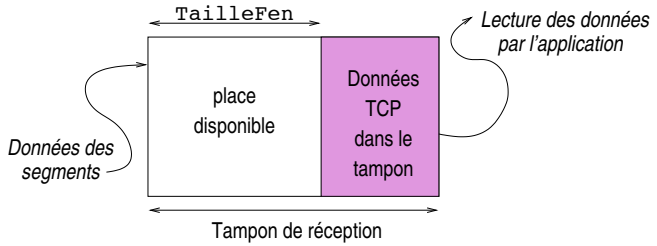
TCP : *Fast Retransmit* (2)



Plan

TCP : Asservissement au récepteur

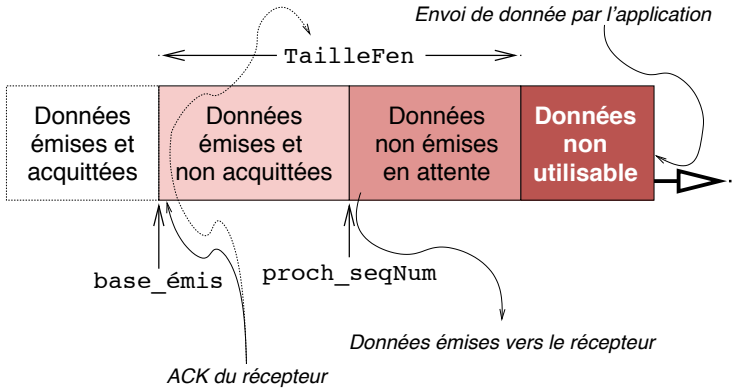
- **contrôle de flux**
  - ✓ l'émetteur ne doit pas dépasser les capacités du récepteur
  - ✓ récupération de la taille de la place disponible du tampon de réception du récepteur :



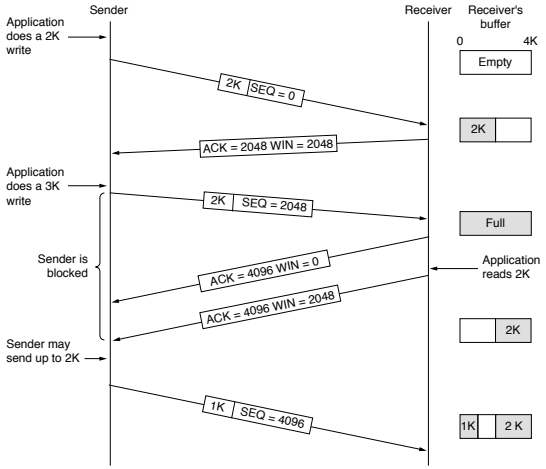
  - ✓ `TailleFen = TailleTampon - DernierOctetRecu + DernierOctetLu`

TCP : Limitation de l'émetteur

Sliding window : l'emetteur limite la transmission de ses données non acquittées



TCP : Contrôle de flux



pictures from TANENBAUM A. S. *Computer Networks 3rd edition*

TCP : temporisation de ré-ouverture de la fenêtre

Persistence timer

- évite que la taille de la fenêtre reste à 0
  - ✓ possible si perte du ACK annonçant une fenêtre non nulle
  - ✓ évité grâce à l'envoi d'un paquet sonde après une temporisation
    - ☞ tempo. initiée à RTT puis double à chaque expiration jusqu'à 60s (puis reste 0 60s)
    - ☞ le paquet sonde est un segment avec 1 octet de données

TCP : Optimisation du contrôle de flux

Send-side silly window syndrome

- Algorithme de Nagle (RFC 896)
  - ✓ agrégation de petits paquets *(nagling)*
  - ✓ attente d'un acquittement ou d'un MSS avant d'envoyer un segment
    - ☞ TELNET : évite d'envoyer un paquet par caractère tapé
    - ☞ désactivable avec l'option TCP_NODELAY des sockets

Receiver silly window syndrome

- Algorithme de Clark
- limiter les annonces de fenêtre trop petites
  - ✓ fermeture de la fenêtre en attendant d'avoir suffisamment de place pour un segment complet

Plan

TCP : exemples d'applications

Les applications suivantes reposent typiquement sur TCP :
- connexion à distance (TELNET, rlogin et ssh)
- transfert de fichiers (FTP, rcp, scp et sftp)
- protocole de routage externe (BGP)
- messageries instantanées (IRC, ICQ, AIM...)
- web (HTTP)
  - ✓ nouvelles applications utilisent HTTP comme service d'accès au réseau
    - ☞ permet de passer les *firewall*

TCP : utilisation spécifiques

TCP doit s'adapter à des flots de qqs **bps** à plusieurs **Gbps** :

- LFN (*Long Fat Network*)
  - ✓ capacité du réseau = **bande passante * délai de propagation**
    - ☞ limitation de taille de la fenêtre (option `WSIZE`, jusqu'à un facteur 2^{14})
    - ☞ rebouclage des numéros de séquence (PAWS, *Protect Against Wrapped Sequence*, utilise l'option `TIMESTAMP`)
    - ☞ acquittements sélectifs pour éviter des retransmissions importantes inutiles (option `SACK`)
  - ✓ satellites
  - ✓ fibres transatlantiques
- réseaux asymétriques (ADSL, Cable)
  - ✓ sous-utilisation du lien rapide

TCP : Interface socket

```
#include <sys/types.h>
#include <sys/socket.h>

# create a descriptor and bind local IP and port
int socket(int domain, int type, int protocol);
#   domain : PF_INET for IPv4 Internet Protocols
#     type : SOCK_STREAM Provides sequenced, reliable, 2-way, connection-based byte streams.
#            An out-of-band data transmission mechanism may be supported.
# protocol : TCP (/etc/protocols)
int bind(int s, struct sockaddr *my_addr, socklen_t addrlen);

# Server : passive queuing mode and connection acceptance
int listen(int s, int backlog);
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

# Client : active connection
int  connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);

# Send and receive data
int send(int s, const void *msg, size_t len, int flags);
int recv(int s, void *buf, size_t len, int flags);

# End : dealocate
int close(int s);
```

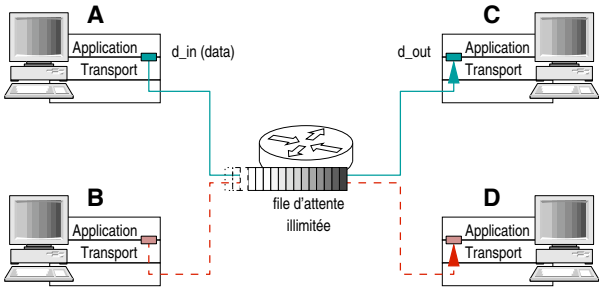

Plan

Contrôle de congestion

Congestion

- trop de flots de données saturent un ou plusieurs éléments du réseau
- différent du contrôle de flux
  - ✓ TCP n'a pas accès à l'intérieur du réseau
- manifestation :
  - ✓ longs délais
    - ☞ attente dans les tampons des routeurs
  - ✓ pertes de paquets
    - ☞ saturation des tampons des routeurs

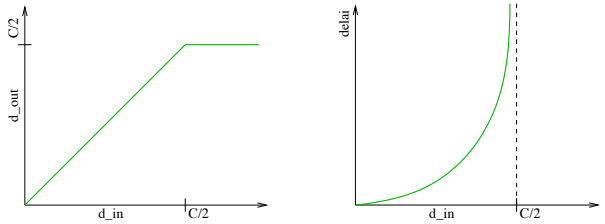
Congestion : scénario 1a



- 2 émetteurs, 2 récepteurs
- 1 routeur
  - ✓ tampons infinis
- pas de retransmission

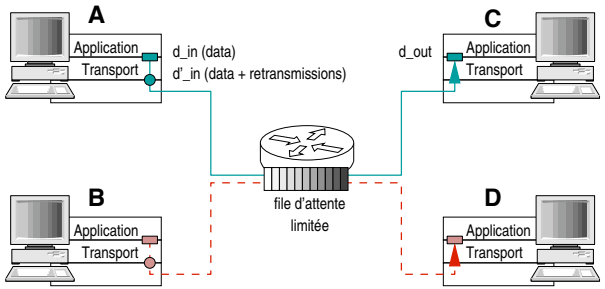
➠ *Que ce passe-t-il quand d_in augmente ?*

Congestion : scénario 1b



- **coût de la congestion :**
  - ✓ débit maximum atteignable
    - ☞ d_in = C/2
  - ✓ delai très élévé proche du maximum
    - ☞ croissance infinie des tampons

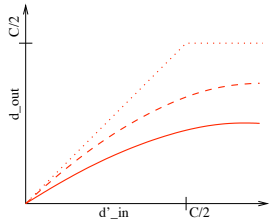
Congestion : scénario 2a



- 2 émetteurs, 2 récepteurs
- 1 routeur
  - ✓ **tampons infinis**
- **retransmission** des segments perdus

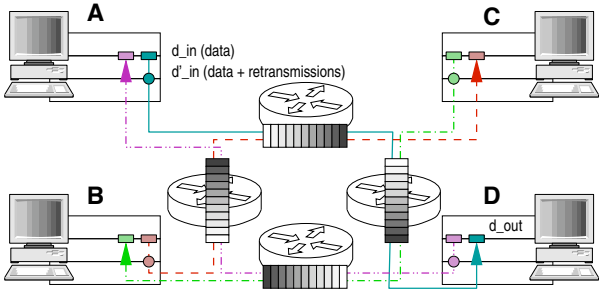
➠ *Que ce passe-t-il quand d'_in augmente ?*

Congestion : scénario 2b



- toujours d_in = d_out *(goodput)*
- coût des retransmissions
  - ✓ **retransmissions utiles** : seulement pour des pertes
    - ☞ d'_in supérieur à d_out
  - ✓ **retransmissions inutiles** : segments en retard
    - ☞ d'_in encore plus supérieur à d_out
- **coût de la congestion** :
  - ✓ beaucoup plus de trafic pour un d_out donné
  - ✓ dupplications de segment inutile

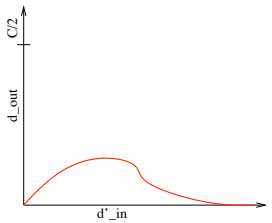
Congestion : scénario 3a



- 4 émetteurs, 4 récepteurs
- 4 routeurs
  - ✓ chemins multi-saut
  - ✓ tampons finis
- retransmission

➠ *Que ce passe-t-il quand d'_in augmente ?*

Congestion : scénario 3b



- **coût supplémentaire de la congestion** :
  - ✓ lors de la perte d'un paquet, toute la capacité amont est gachée

Contrôle de congestion

Deux approches :

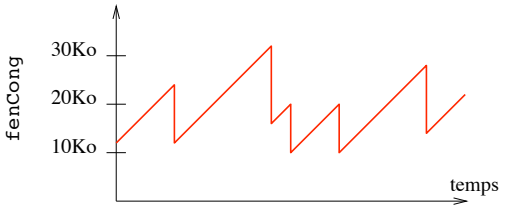
- contrôle de congestion géré par le **réseau**
  - ✓ les routeurs informent les extrémités
    - ☞ bit d'indication de la congestion (SNA, DECbit, ATM, TCP/IP ECN...)
    - ☞ indication explicte du débit disponible (ATM ABR, TCP/IP RSVP + IntServ...)
- contrôle de congestion aux **extrémités** *(end-to-end)*
  - ✓ aucune indication explicite du réseau
  - ✓ **inférence** à partir des observations faites aux extrémités
    - ☞ pertes
    - ☞ délais
  - ✓ **approche choisie dans TCP**

Plan

TCP : Contrôle AIMD

Additive Increase, Multiplicative Decrease

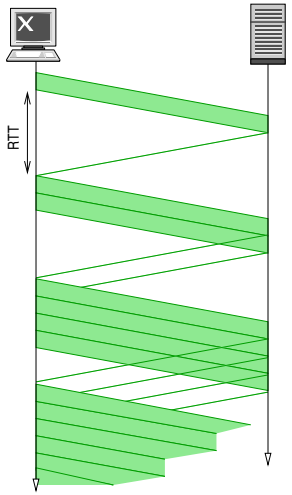
- augmentation progressive du débit de transmission (`fenCong`) tant qu'il n'y a pas de perte
  - ✓ *Additive Increase*
    - ☞ augmenter `fenCong` de 1 MSS à chaque RTT tant qu'il n'y a pas de perte détectée
  - ✓ *Multiplicative Decrease*
    - ☞ diviser `fenCong` par 2 après une perte
  - ✓ comportement en dent de scie :



TCP : Contrôle de congestion

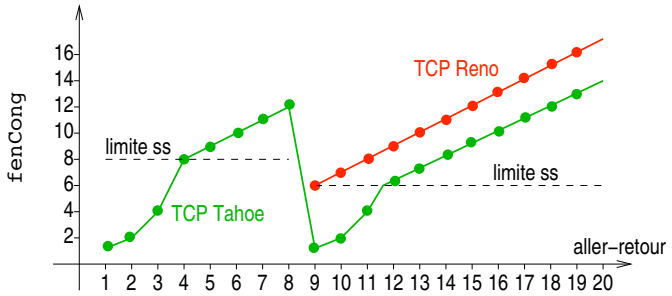
- basé sur la limitation de l'émission de l'émetteur
  - ✓ `dernierOctetEmis - dernierOctetAcq` $\leq$ `fenCong`
  - ✓ approximation du débit :
    - ☞ $d_{TCP} = \frac{\texttt{fenCong}}{RTT}$
- `fenCong` = fonction dynamique de la congestion perçue
  - ✓ perception de la congestion par le récepteur :
    - ☞ expiration de temporisation (RTO)
    - ☞ triple ACK
  - ✓ 3 mécanismes :
    - ☞ AIMD
    - ☞ *Slow Start*
    - ☞ prudence après l'expiration d'une temporisation

TCP : Slow Start



- démarre lentement *(slow start)*
  - ✓ mais croit très vite ! !
- au démarrage de la connexion
  - ✓ `fenCong` = 1 MSS
    - ☞ $d_{init} = \frac{MSS}{RTT}$
  - ✓ croissance exponentielle jusqu'à la première perte
    - ☞ `fenCong` double / RTT
    - ☞ implémenté par : `fenCong` ++ / ACK
    - ☞ $d_{potentiel} \gg \frac{MSS}{RTT}$

TCP : Optimisation



- passage de la croissance exponentielle à lineaire
  - ✓ `fenCong` $\geq$ ancienne valeur de `fenCong` juste avant la perte
    - ☞ implémenté par une limite variable : `limiteSS` = $\texttt{fenCong}_{avant\ la\ derniere\ perte}/2$

`limiteSS` est précisément calculé avec les segments non acquittés *(flightsize)*/2

TCP : Inférence des pertes

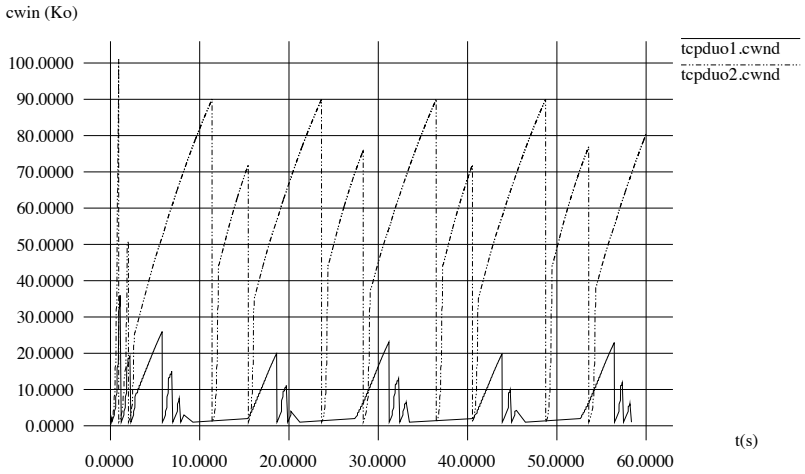
Les ACK dupliqués sont moins graves que les expirations de temporisation

- suite **3 ACK dupliqués** :
  - ✓ indique que le réseau continue à transmettre des segments
    - ☞ `fenCong` divisé par 2
    - ☞ `fenCong` croit ensuite linéairement
- suite **expiration temporisation** :
  - ✓ indique que le réseau se bloque
    - ☞ `fenCong` = 1 MSS
    - ☞ *Slow Start* (croissance exponentielle)
    - ☞ à `limiteSS` = `fenCong`/2 (croissance linéaire)

Contrôle de congestion TCP : synthèse

- quand `fenCong` $<$ `limiteSS` :
  - ✓ émetteur en phase *Slow Start*
  - ✓ `fenCong` croit exponentiellement
- quand `fenCong` $\geq$ `limiteSS` :
  - ✓ émetteur en phase *Congestion Avoidance*
  - ✓ `fenCong` croit linéairement
- quand 3 ACK dupliqués apparaissent :
  - ✓ `limiteSS` = dernière `fenCong` / 2
  - ✓ `fenCong` = `limiteSS`
- quand la temporisation expire :
  - ✓ `limiteSS` = dernière `fenCong` / 2
  - ✓ `fenCong` = 1 MSS

TCP : équité entre flots ?



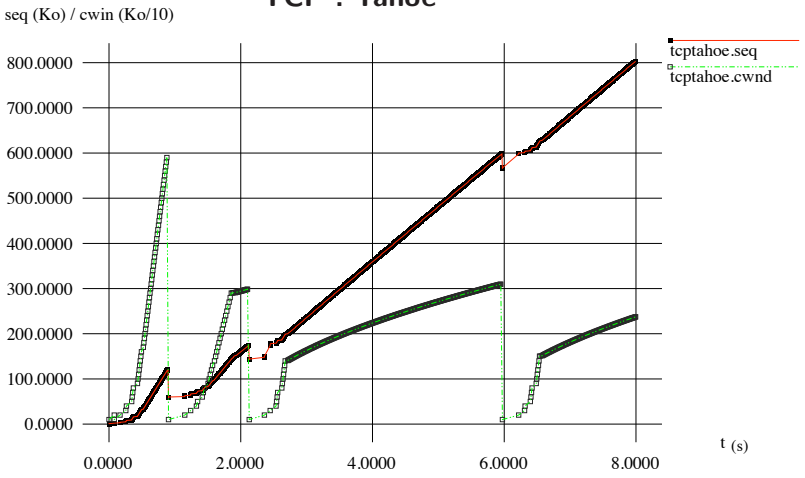
- oscillation de deux flots en phase de congestion

Implémentations

A trip to Nevada :

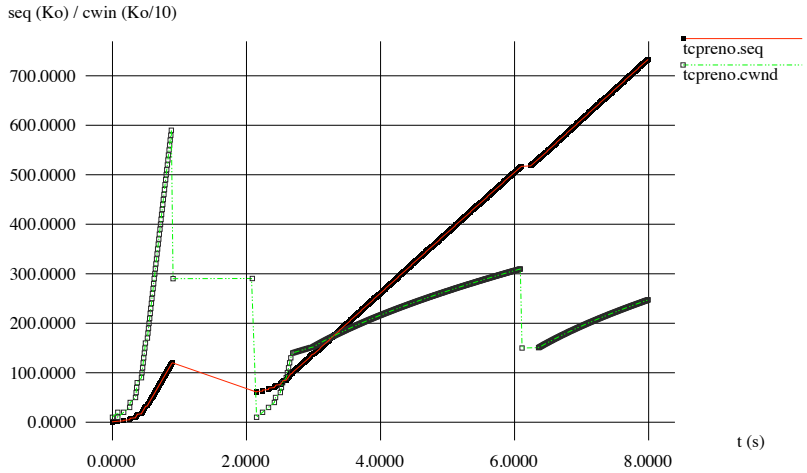
- **TCP Tahoe** 1988
  - ✓ *slow start* + *congestion avoidance* + *multiplicative decrease*
  - ✓ + **fast retransmit** (déclenche la retransmission d'un segment après trois acquittements dupliqués, avant l'expiration de la temporisation)
  - ✓ décrit précédement... problème quand juste un segment est perdu
- **TCP Reno** 1990 (RFC 2581)
  - ✓ idem TCP Tahoe
  - ✓ + **fast recovery** (pas de *slow start* après un *fast retransmit*)
- **TCP newReno** 1996 (RFC 3782)
  - ✓ idem TCP Reno
  - ✓ + pas de *slow start* à la première congestion et ajustement de `fenCong`
  - ✓ + SACK (RFC 2018)
- **TCP Vegas...**
  - ✓ évite la congestion en **anticipant** les pertes
  - ✓ réduction du débit en fonction des variations du RTT

TCP : Tahoe



- *slow start* + *congestion avoidance* + *multiplicative decrease*
- + **fast retransmit** : vers quel débit converge-t-on ?

TCP : Reno



- **fast recovery** (pas de *slow start* après un *fast retransmit*)

Fin

Document réalisé avec LaTeX.
Classe de document `foils`.
Dessins réalisés avec `xfig`.

Olivier Fourmaux, `olivier.fourmaux@upmc.fr`
`http://www-rp.lip6.fr/~fourmaux`

Ce document est disponible en format PDF sur le site :
`http://www-master.ufr-info-p6.jussieu.fr/`