# Protocoles de transport - Plan

1. Rôle du transport
2. Le protocole UDP
3. Le protocole TCP



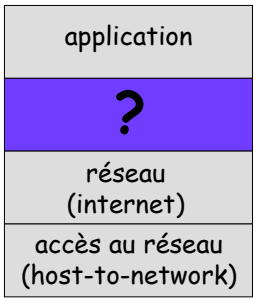
# Protocoles de transport - Plan

1. **Rôle du transport**
2. Le protocole UDP
3. Le protocole TCP



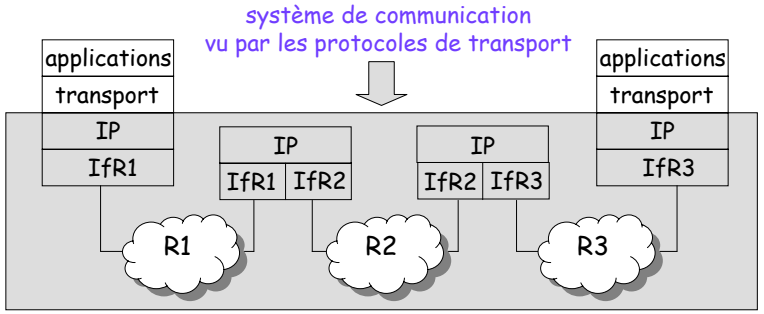
# Problématique

- ✓ différentes technologies possibles pour connecter un ensemble de machines
  - LAN
  - WAN
  - inter-réseaux
- ↳ service de remise de paquets de machine à machine
- ✓ comment passer de ce service à un canal de communication de processus à processus ?






# Le système de communication




- ✓ service fourni par IP
  - routage à travers une interconnexion de réseaux
  - fragmentation/réassemblage
  - service non connecté, *best effort*

# Challenges

- ✓ prendre en compte le service fourni par la couche réseau
  - pertes de paquets
  - déséquencements
  - duplications
  - erreurs
  - MTU
  - temps de traversée imprévisibles
  - …
- ✓ prendre en compte les besoins des applications
  - garantie de remise des messages
  - séquencement
  - absence de duplications
  - absence de messages erronés
  - messages de lg quelconque
  - synchronisation entre l'émetteur et le récepteur
  - contrôle de flux par le récepteur sur l'émetteur
  - support de plusieurs applications sur le même hôte
  - …




# Rôle de la couche transport

- ✓ transformer les propriétés pas toujours désirables du réseau en un service de haut niveau souhaité par les applications
- ✓ plusieurs déclinaisons de protocoles de transport
  - UDP
  - TCP
  - …




# Protocoles de transport - Plan

1. Rôle du transport
2. Le protocole UDP
   - le (dé)multiplexage
   - la notion de port
   - le datagramme UDP
   - autres fonctions ?
3. Le protocole TCP




# Le protocole UDP

- ✓ User Datagram Protocol
- ✓ se contente d'étendre
  - le service de remise d'hôte à hôte à
  - un service de remise de processus à processus
- ✓ Problème
  - plusieurs applications peuvent tourner simultanément sur un même hôte
  - ↳ il faut donc pouvoir les identifier de façon non ambiguë
  - ↳ introduction d'un niveau supplémentaire de multiplexage
    - cf. le champ type d'Ethernet qui identifie à qui doit être délivré le contenu du champ de données de la trame
    - cf. le champ protocol de IP qui identifie à qui doit être délivré le contenu du champ de données du datagramme


# Multiplexage/démultiplexage

- ✓ Problème
  - comment identifier un processus (une application) ?
- ✓ solution 0
  - on peut identifier directement un processus sur une machine par son *pid* (*process identifier*)
  - ☹ malheureusement possible que si l'OS est un OS distribué "fermé" qui alloue un *pid* unique à chaque processus



# Multiplexage/démultiplexage

- ✓ solution
  - on peut identifier indirectement un processus par une référence abstraite (*abstract locater*) appelée port
    - un processus source envoie un message sur son port
    - un processus destinataire reçoit le message sur son port
    - port ~ boîte aux lettres
- ✓ réalisation de la fonction de (dé)multiplexage
  - champ *port* (de la) *source* du message
  - champ *port* (de la) *destination* du message



# Multiplexage/démultiplexage

- ✓ champs *port* codés sur 16 bits
  - 65 535 valeurs différentes ➔ insuffisant pour identifier tous les processus de tous les hôtes de l'Internet
  - les ports n'ont pas une signification globale
    - signification restreinte à un hôte
    - ↳ un processus est identifié par son port sur une machine donnée
  - ↳ clé de démultiplexage de UDP = (port, hôte)



# Multiplexage/démultiplexage

- ✓ comment un processus connaît-il le port de celui à qui il souhaite envoyer un message ?
  - modèle de communication client/serveur
  - une fois que le client a contacté le serveur, le serveur connaît le port du client
- ✓ comment le client connaît-il le port du serveur ?
- ✓ le serveur accepte des messages sur un port connu de tous (*well-known port*)
  - ex : pompiers : 18, police : 17, SAMU : 15
  - ex : DNS : 53, Telnet : 23, http : 80

# Numéros de port

- ✓ 3 catégories
  - ports well-known : de 0 à 1023
    - alloués par l'IANA
    - sur la plupart des systèmes, ne peuvent être utilisés que par des processus système (ou root) ou des programmes exécutés par des utilisateurs privilégiés
  - ports registered : de 1024 à 49 151
    - listés par l'IANA
    - sur la plupart des systèmes, peuvent être utilisés par des processus utilisateur ordinaires ou des programmes exécutés par des utilisateurs ordinaires
  - ports dynamic/private : de 49 152 à 65 535
    - alloués dynamiquement

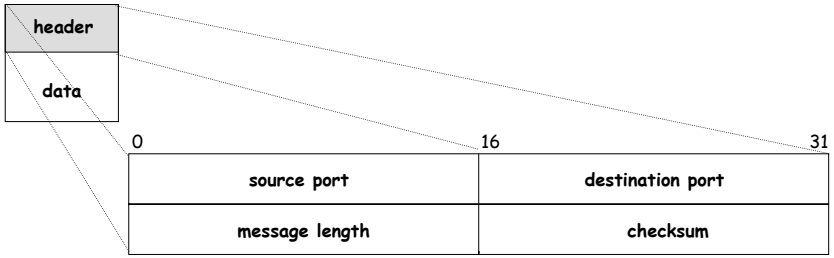


# Les ports *well-known*

| No port | Mot-clé | Description |
|---|---|---|
| 7 | ECHO | Echo |
| 11 | USERS | Active Users |
| 13 | DAYTIME | Daytime |
| 37 | TIME | Time |
| 42 | NAMESERVER | Host Name Server |
| 53 | DOMAIN | Domain Name Server |
| 67 | BOOTPS | Boot protocol server |
| 68 | BOOTPC | Boot protocol client |
| 69 | TFTP | Trivial File Transfert Protocol |
| 123 | NTP | Network Time Protocol |
| 161 | SNMP | Simple Network Management Protocol |



# Le datagramme UDP




- ✓ fonctionnalités autres que le (dé)multiplexage ?
  - pas de contrôle de flux
  - pas de fiabilité
  - détection d'erreurs ?
  - fragmentation ?



# Le checksum UDP

- ✓ calcul optionnel avec IPv4, obligatoire avec IPv6
- ✓ portée
  - l'en-tête UDP
  - le champ de données UDP
  - un *pseudo-header*
    - champ IP *protocol* (8 bits cadrés à droite sur 16 bits)
    - champ IP *@source* (32 bits)
    - champ IP *@dest.* (32 bits)
    - champ UDP *length* (16 bits)

↳ UDP est indissociable de IP !

# Le checksum UDP

- ✓ algorithme de calcul
  - principe
    - le champ *checksum* est initialement mis à 0
    - la suite à protéger est considérée comme une suite de mots de 16 bits
    - les mots de 16 bits sont additionnés un à un, modulo 65 535
    - le checksum est le complément à 1 (inverse bit à bit) de la somme trouvée
    - ↳ le récepteur fait la somme modulo 65 535 de tous les mots concernés et vérifie qu'il obtient FF FF ou 00 00
  - ☺ implémentation logicielle simple
  - ☹ moins puissant qu'un CRC



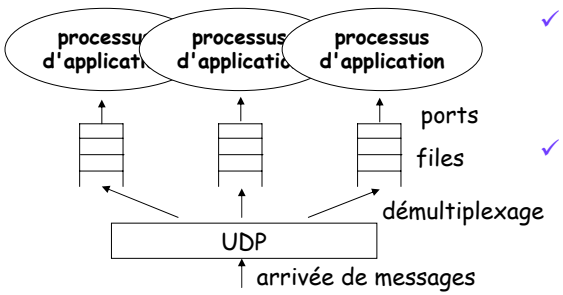
# Fragmentation

- ✓ en théorie
  - les messages UDP peuvent être fragmentés par IP
- ✓ en pratique
  - la plupart des applications utilisant UDP limitent leurs messages à 512 octets
  - ↳ pas de fragmentation
  - ↳ pas de risque de message incomplet



# Implémentation des ports

- ✓ port = référence *abstraite*
  - ↳ l'implémentation diffère d'un OS à l'autre !
- ✓ en général, un port = une file de messages




- ✓ quand un msg arrive, UDP l'insère en fin de file
  - si la file est pleine, le msg est rejeté
- ✓ quand le processus veut recevoir un msg, il le retire de la tête de la file
  - si la file est vide, le processus se bloque jusqu'à ce qu'un msg soit disponible



# Protocoles de transport - Plan

1. Rôle du transport
2. Le protocole UDP
3. Le protocole TCP
   - transport vs. liaison
   - flux d'octets et segments
   - le segment TCP
   - l'établissement de connexion
   - la libération de connexion
   - le contrôle de flux
   - le contrôle d'erreur

# Le protocole TCP

- ✓ Transmission Control Protocol
- ✓ offre un service de remise
  - en mode connecté
  - fiable
  - full-duplex
  - de flux d'octets
- ✓ met en œuvre des mécanismes de
  - (dé)multiplexage
  - gestion de connexions
  - contrôle d'erreur
  - contrôle de flux
  - contrôle de congestion




# Comparaison avec une liaison de données

1. gestion des connexions
   - la liaison est bâtie sur un canal physique reliant toujours les 2 mêmes ETTD
   - TCP supporte des connexions entre 2 processus s'exécutant sur des hôtes quelconques de l'Internet
   - ↳ phase d'établissement plus complexe
2. le RTT (*Round Trip Time*) est
   - pratiquement constant sur une liaison
   - varie en fonction de l'heure de la connexion et de la "distance" séparant les 2 hôtes
   - ↳ dimensionnement du temporisateur de retransmission




# Comparaison avec une liaison de données

3. les unités de données
   - ne se doublent pas sur le support de transmission
   - peuvent se doubler et peuvent être retardés de façon imprévisible dans le réseau
   - ↳ TCP doit prévoir le cas de (très) vieux paquets réapparaissent
4. les buffers de réception
   - sont propres à *la* liaison
   - sont partagés entre toutes les connexions ouvertes
   - ↳ TCP doit adapter le mécanisme de contrôle de flux




# Comparaison avec une liaison de données

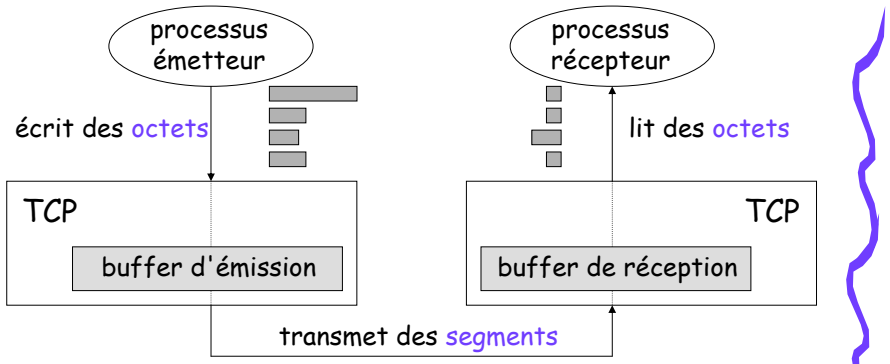
5. une congestion
   - du lien sur une liaison de données ne peut pas se produire sans que l'émetteur ne s'en rende compte
   - du réseau peut se produire
   - ↳ TCP va mettre en œuvre un contrôle de congestion


# Byte-oriented

- ✓ TCP est orienté flux d'octets
  - le processus émetteur "écrit" des octets sur la connexion TCP
  - le processus récepteur "lit" des octets sur la connexion TCP
- ✓ TCP ne transmet pas d'octets individuels
  - en émission
    - TCP bufferise les octets jusqu'à en avoir un nombre raisonnable
    - TCP fabrique un segment et l'envoie
  - en réception
    - TCP vide le contenu du segment reçu sans un buffer de réception
    - le processus destinataire vient y lire les octets à sa guise



# Byte-oriented





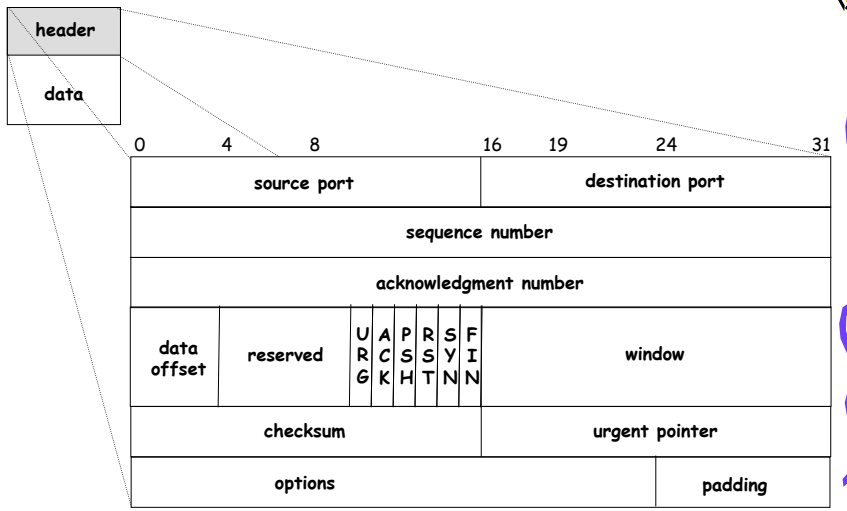

# Construction d'un segment

- ✓ Quand est-ce que TCP décide d'envoyer un segment ?

1. il a MSS (MaximumSize Segment) octets de données à envoyer
   - MSS = MTU – en-tête IP – en-tête TCP
2. le processus lui demande explicitement
   - fonction *push*
3. le temporisateur expire
   - pour éviter d'attendre trop longtemps MSS octets



# Le segment TCP




options

# Les champs de l'en-tête TCP

- source port : identifie le processus source sur la machine source
- destination port : identifie le processus destinataire sur la machine destinataire
- sequence number : N° du 1er octet de données du segment (sauf si SYN=1 : ISN)
- acknowledgment number : acquitte tous les octets de données de N° strictement inférieur
- data offset : lg de l'en-tête en mots de 32 bits
- reserved : 6 bits à 0
- URG : mis à 1 pour signaler la présence de données urgentes
- ACK : mis à 1 pour indiquer que le champ acknowledment number est significatif
- PSH : mis à 1 pour signaler la fin d'un message logique (push)
- RST : mis à 1 pour réinitialiser la connexion (panne, incident, segment invalide)
- SYN : mis à 1 pour l'établissement de la connexion
- FIN : mis à 1 pour fermer le flux de données dans un sens
- window : # d'octets de données que le destinataire du segment pourra émettre
- checksum : obligatoire, calculé sur la totalité du segment et sur le pseudo en-tête
- urgent pointer : pointe sur la fin (comprise) des données urgentes
- options : MSS, …
- padding : alignement de l'en-tête sur 32 bits



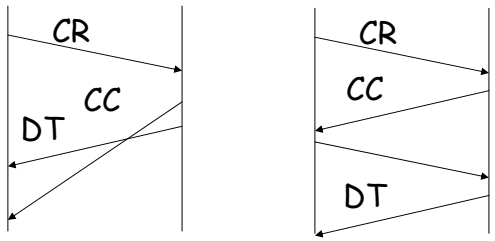
# Les ports *well-known*

| No port | Mot-clé | Description |
|---|---|---|
| 20 | FTP-DATA | File Transfer [Default Data] |
| 21 | FTP | File Transfer [Control] |
| 23 | TELNET | Telnet |
| 25 | SMTP | Simple Mail Transfer |
| 37 | TIME | Time |
| 42 | NAMESERVER | Host Name Server |
| 43 | NICNAME | Who Is |
| 53 | DOMAIN | Domain Name Server |
| 79 | FINGER | Finger |
| 80 | HTTP | WWW |
| 110 | POP3 | Post Office Protocol - Version 3 |
| 111 | SUNRPC | SUN Remote Procedure Call |



# L'établissement de connexion

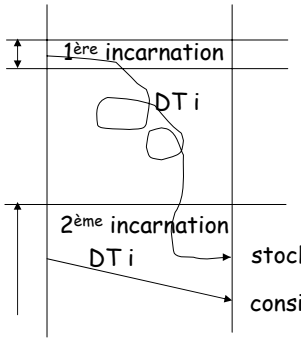
- en 3 phases (*three-way handshake*)






# L'établissement de connexion

- avec échange des ISN (Initial Sequence Number)

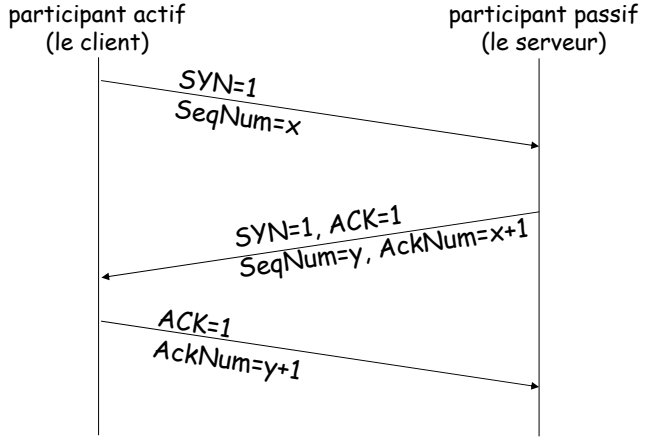



stockée pour reséquencement puis délivrée

considérée comme retransmission

# L'établissement de connexion

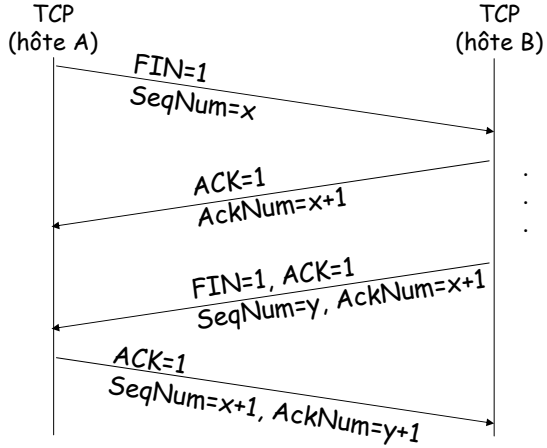
- ✓ les 3 segments échangés






# La libération de connexion

- ✓ les 2 sens de transmission sont fermés séparément





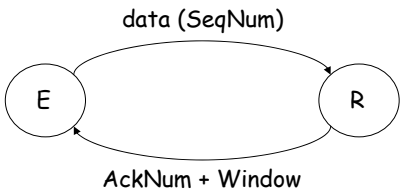

# Transfert de données

- ✓ TCP assure un transfert de bout en bout fiable
  - ↳ contrôle de flux
  - ↳ contrôle d'erreur



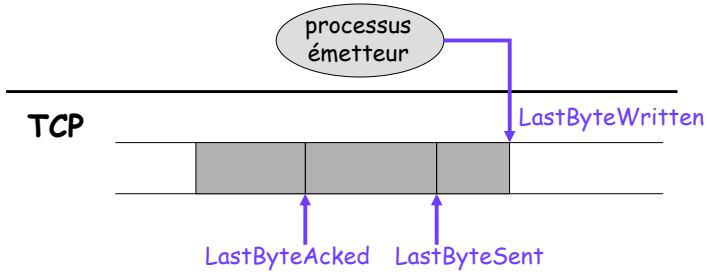
# Le contrôle de flux

- ✓ une entité réceptrice TCP dispose de ressources (buffers) en nombre limité
- ✓ la taille de la fenêtre reflète la disponibilité des buffers de réception
- ✓ une entité TCP gère un nombre de connexions variable
  - ↳ fenêtre dynamique
    - progression de la fenêtre par acquittement et crédit
    - champ *window*
    - indique le # d'octets de données que l'entité est prête à recevoir



# Buffer d'émission

- ✓ en émission, un buffer stocke
  - les données envoyées et en attente d'acquittement
  - les données passées par le processus émetteur mais non encore émises





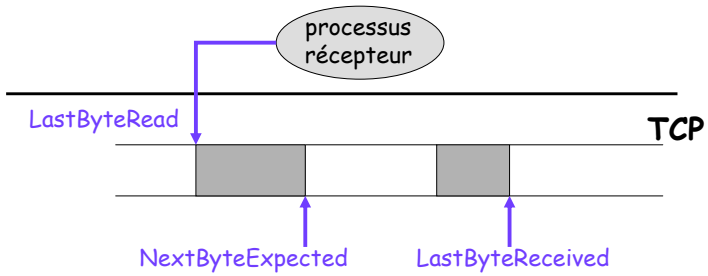

# Buffer d'émission

- remarque : on a toujours
  - LastByteAcked <= LastByteSent <= LastByteWritten
- TCP vérifie à tout moment que
  - LastByteSent – LastByteAcked <= AdvertisedWindow
- TCP calcule une fenêtre effective
  - EffectiveWindow <- AdvertisedWindow – (LastByteSend – LastByteAcked)
- en parallèle, TCP doit s'assurer que le processus émetteur ne sature pas on buffer
  - si le processus veut écrire y octets et que (LastByteWritten – LastByteAcked) + y > MaxSendBuffer TCP bloque l'écriture



# Buffer de réception

- ✓ en réception, un buffer stocke
  - les données dans l'ordre non encore lues par le processus récepteur
  - les données déséquencées






# Buffer de réception

- remarque : on a toujours
  - LastByteRead < NextByteExpected <= LastByteReceived +1
- TCP vérifie à tout moment que
  - MaxRcvBuffer >= LastByteReceived – LastByteRead
- TCP communique une fenêtre mise à jour
  - AdvertisedWindow <- MaxRecvBuffer – (LastByteReceived – LastByteRead)
- au fur et à mesure que les données arrivent
  - TCP les acquitte si tous les octets précédents ont été reçus
  - LastByteReceived glisse vers la droite ➔ rétrécissement possible de la fenêtre

# Réouverture de fenêtre

- ✓ Problème
  - la fenêtre peut avoir été fermée par le récepteur
  - un ACK ne peut être envoyé que sur réception de données (approche *smart sender/dumb receiver*)
  - ↳ comment l'émetteur peut-il se rendre compte de la réouverture de la fenêtre ?
- ✓ solution
  - lorsque l'émetteur reçoit une AdvertisedWindow à 0, il envoie périodiquement un segment avec 1 octet de données pour provoquer l'envoi d'un ACK



# Contrôle d'erreur

- ✓ repose sur
  - le champ Checksum
  - le champ SequenceNumber
  - des acquittements positifs (*dumb receiver* ➔ pas d'acquittements négatifs)
  - un temporisateur de retransmission
  - des retransmissions
- ✓ RTT variable
- ↳ dimensionnement dynamique du temporisateur



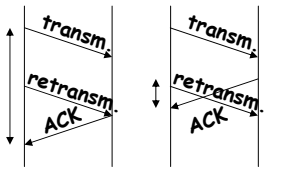
# Dimensionnement du temporisateur

- ✓ algorithme initial
  1. TCP calcule une estimation du RTT par une moyenne pondérée entre l'estimation précédemment calculée et le dernier échantillon mesuré du RTT
     - EstimatedRTT <- $\alpha$ EstimatedRTT + (1 – $\alpha$) SampleRTT
     - SampleRTT est obtenu en mesurant le délai séparant l'émission d'un segment de la réception de son acquittement
     - 0,8 < $\alpha$ < 0,9
  2. TCP calcule la valeur du temporisateur
     - TimeOut <- min (UBOUND, max (LBOUND, $\beta$ EstimatedRTT))
     - UBOUND est une limite supérieure sur le timer (p.e. 60 s)
     - LBOUND est une limite inférieure sur le timer (p.e. 1 s)
     - 1,3 <= $\beta$ <= 2



# Dimensionnement du temporisateur

- ✓ Problème
  - un ACK n'acquitte pas une transmission, mais une réception




SampleRTT trop grand — SampleRTT trop petit

- ✓ Algorithme de Karn-Partridge
  - ne mesurer SampleRTT que pour les segments envoyés une seule fois
  - calcul de TimeOut
    - à chaque retransmission, doubler la valeur de TimeOut, jusqu'à ce que la transmission réussisse
    - pour les segments suivants, conserver la valeur du TimeOut, jusqu'à ce qu'un segment soit acquitté du 1er coup
    - recalculer TimeOut à partir de EstimatedRTT