Cours Architectures des Réseaux
L2 Informatique

Protocoles de niveau transport

Lila Boukhatem
LISN : Laboratoire interdisciplinaire des sciences du numérique
Université Paris Saclay
Lila.Boukhatem@universite-paris-saclay.fr

1

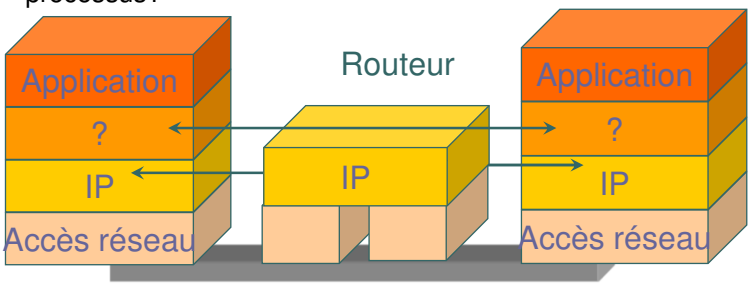
plan

- Rôle de la couche transport
- Le protocole UDP
- Le protocole TCP
- conclusion

2

Introduction

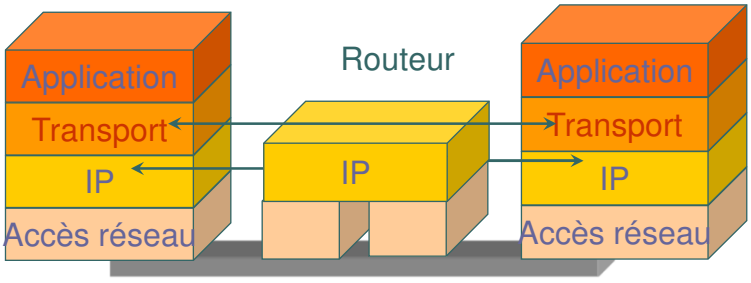
- IP rend un service de remise de paquets de machine à machine
  - Routage à travers des réseaux interconnectés
  - Fragmentation/réassemblage
  - Service en mode non connecté – Best effort
- Problème
  - Comment passer vers une communication processus à processus?



3

Introduction

- Couche de transport
  - Fournit un service de communication entre processus applicatifs se trouvant sur des machines d'extrémité
  - Communication de type « logique »
    - Logique : car du point de vue de l'applicatif, les machines sont en communication directe même si elles sont distantes et séparées par des routeurs
  - Transport de « bout-en-bout »



4

Les difficultés

- Prise en compte du service fourni par la connexion réseau
  - Pertes
  - Déséquencement
  - Erreurs
  - Temps de traversée imprévisibles
  - Multiplicité des réseaux utilisés pour l'acheminement - MTU
- Prise en compte des besoins de l'application
  - Garantie de remise des paquets
  - Séquencement
  - Garantie de remise sans erreur
  - Messages de longueur quelconque
  - Synchronisation émetteur/récepteur
  - Support de plusieurs applis sur la même machine

5

Le protocole de transport - Rôle

- **Communication entre processus**
- **Bout en bout**
- **Services offerts**
  - Corriger les défauts du service réseau
    - Transport fiable
    - Délivrance des données en séquence
    - Contrôle de flux et contrôle de congestion
    - Multiplexage et démultiplexage
  - Compte rendu sur les performances de la communication

6

Les protocoles de transport

- Plusieurs protocoles de transport
  - TCP (Transmission Control Protocol)
    - Service fiable et orienté connexion
  - UDP (User Datagram Protocol)
    - Service non fiable et orienté non connexion
  - RTP/RTCP
    - Service orienté voix-vidéo temps réél

7

plan

- Rôle de la couche transport
- Le protocole UDP
  - Le (de)muliplexage
  - Les ports
  - Le datagramme UDP
- Le protocole TCP
- conclusion

8

Le protocole UDP

- User Datagram Protocol (rfc 768)
- Se contente d'étendre
  - le service de remise d'hôte à hôte à
  - un service de remise de processus à processus
- Caractéristiques
  - Protocole simple
  - Service non fiable
    - L'arrivée des messages ainsi que le séquencement ne sont pas garantis
  - Mode non connecté
  - Absence de contrôle de flux/ congestion
  - Multiplexage et démultiplexage
  - Protocole rapide (pas de délai connexion, pas de contrôle de congestion : émission rapide)

9

Multiplexage/démultiplexage

- constat
  - plusieurs applications s'exécutent simultanément sur une machine
  - Besoin :
    - les identifier de façon unique et non ambiguë
    - introduire un niveau supplémentaire de multiplexage
      - Niveau 2 : Ethernet → le champ type identifie à qui doit être délivré le contenu du champ de données de la trame
      - Niveau 3 : IP → le champ protocol de IP identifie à qui doit être délivré le contenu du champ de données du paquet

10

Multiplexage/démultiplexage

- Problème
  comment identifier un processus (une application) ?
- solution
  - on peut identifier 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 grâce au
  - champ *port* (de la) *source* du message
  - champ *port* (de la) *destination* du message

11

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)

12

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

13

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

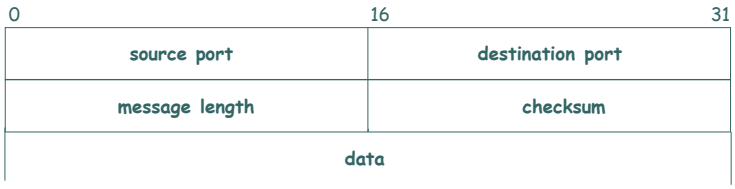
14

Quelques ports réservés

No port	Mot-clé	Description
13	DAYTIME	Daytime
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
67	BOOTPS	Boot protocol server
68	BOOTPC	Boot protocol client
69	TFTP	Trivial File Transfert Protocol
80	www-http	World Wide Web HTTP
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol

15

Le datagramme UDP



- Message length : longueur totale (en-tête + données)
- Checksum?

16

Le checksum UDP

- calcul optionnel avec IPv4 (vaut 0 si pas utilisé), 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 !

17

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
    - Bourrage pour alignement sur 16 bits (obtenir un nombre pair d'octets)
    - 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

18

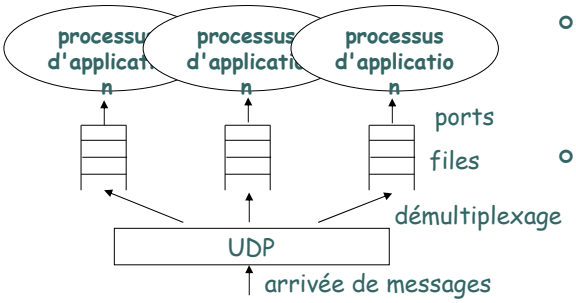
Fragmentation

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

19

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

20

plan

- Rôle de la couche transport
- Le protocole UDP
- Le protocole TCP
  - Principes
  - Gestion des connexions
  - Fiabilité
  - Contrôle de flux
  - Contrôle de congestion
  - Format du datagramme
- conclusion

21

TCP (Transmission Control Protocol)

- 90% du trafic d'Internet
- Principes
  - Transfert fiable d'un flot d'octets entre processus applicatifs distants
    - Contrôle de perte
    - Contrôle de flux (interaction entre machines d'extrémité)
    - Contrôle de congestion (interaction machines hôte/réseau)
  - Orienté flux d'octets
    - L'application lit et écrit des octets
    - TCP envoie des segments
  - Mode connecté
  - Point à point et bidirectionnel (full duplex)

22

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

23

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 qui 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

24

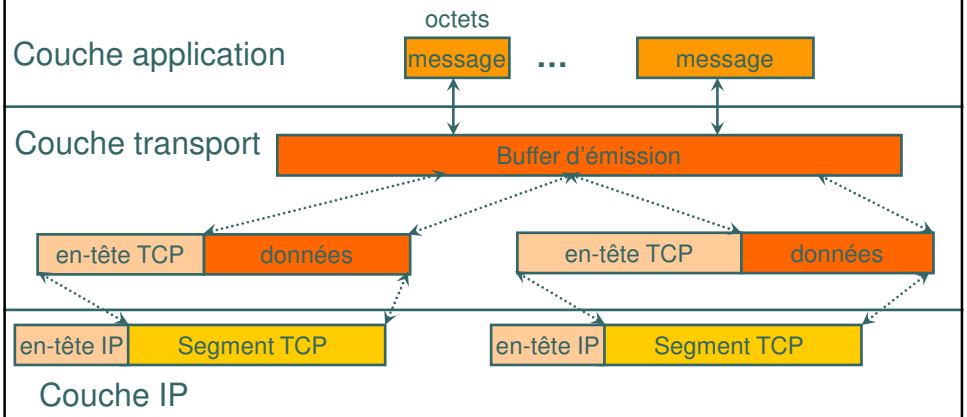
Comparaison avec une liaison de données

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

25

Le protocole TCP - Principes

- Segmentation et segments TCP
  - TCP gère deux tampons : en émission et en réception
  - Les octets sont envoyés dans des segments de taille maximale MSS (Maximum Segment Size)



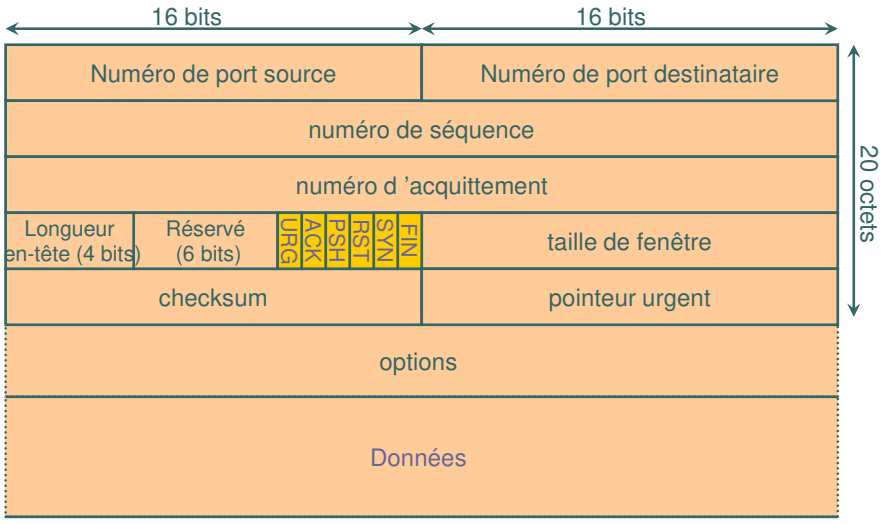
26

Construction d'un segment

- 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 dans un buffer de réception
    - le processus destinataire vient y lire les octets à sa guise
- Quand TCP décide-t-il d'envoyer un segment?
  - Il dispose de MSS octets de données à envoyer
    - MSS = MTU – en-tête IP – en-tête TCP
  - Le processus lui demande explicitement
    - Fonction *push*
  - Le temporisateur expire
    - Pour éviter d'attendre trop longtemps MSS octets

27

Le segment TCP



28

Le segment TCP

- Numéro de port source
  - Identification de l'application source
- Numéro de port destinataire
  - Identification de l'application destinataire
- Quadruplet (IP1, port1, IP2, port2) identification de la connexion
  - Port + @IP (source ou destinataire) = socket sous Unix
- Numéro de séquence
  - Identification des octets envoyés par la source
    - Numéro du premier des octets de données contenues dans le segment
  - Pour un segment ayant SYN=1 : il s'agit de l'ISN
- Numéro d'acquittement
  - Identification des octets reçus par le destinataire
  - Numéro du prochain octet attendu
- Longueur de l'en-tête
  - Indique la taille de l'en-tête en mots de 32 bits
  - 20 octets sans option

29

Le protocole TCP – Format du segment TCP

- Taille de la fenêtre
  - Nombre d'octets que le récepteur peut recevoir (fenêtre de réception)
  - Utilisé pour le contrôle de flux
- Le Checksum
  - Détection des erreurs bit sur le segment : pseudo header + TCP header + data
- Pointeur Urgent
  - Détermine le dernier octet des données urgentes contenues dans le champ *données*
- Option
  - Fait partie de l'en-tête
  - Longueur variable
  - Exemple : MSS (Maximum Segment Size)
- Réservé
  - Réservé pour une utilisation future

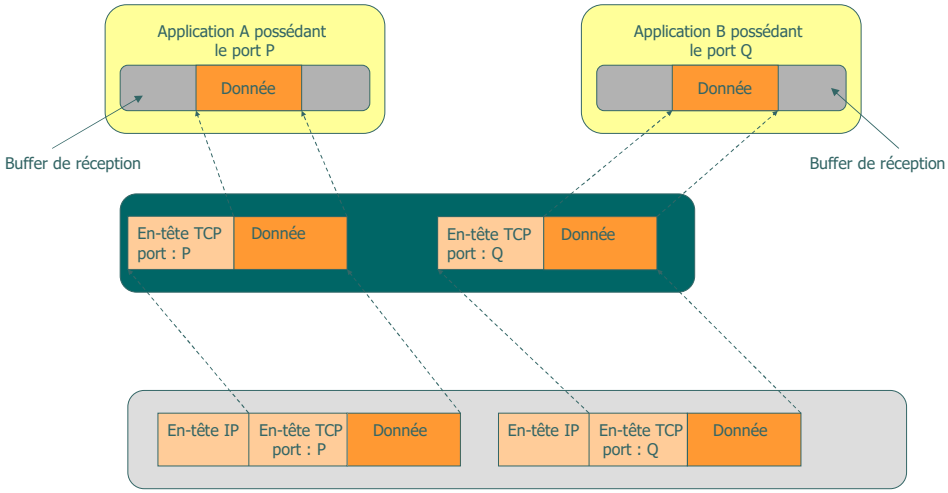
30

Le protocole TCP – Format du segment TCP

- Flags (drapeaux)
  - Codé sur 1 bit (0=inactif, 1=actif)
  - URG (URGent): segment avec données urgentes
    - Les données entre le 1er octet des données et la valeur du champ Pointeur Urgent sont urgentes
  - ACK (ACKnoledgment) : La valeur du champ « acquittement » peut être prise en compte (valide)
  - PSH (PuSH) : données à remettre immédiatement à l'application
    - Pour optimiser la transmission, TCP attend que les buffers soient pleins
    - PSH sert à demander l'émission et la réception immédiates
  - SYN (SYNchronisation) : segment d'ouverture de connexion
  - RST (ReSet) : fermeture immédiate pour cause d'anomalie
  - FIN : segment de fin de connexion (plus de données à émettre)

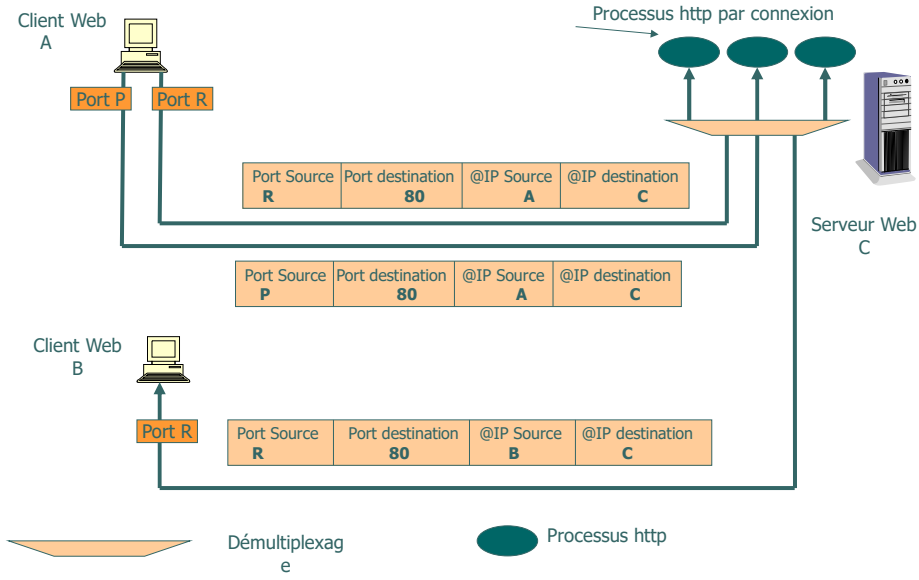
31

Le protocole TCP - Principes - Multiplexage/Démultiplexage



32

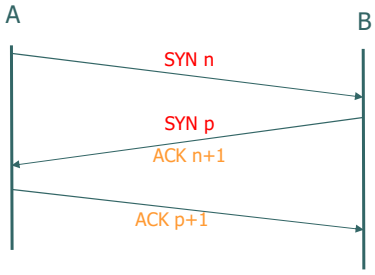
Le protocole TCP - Principes - Multiplexage/Démultiplexage



33

Connexions TCP – Ouverture de connexion

- Ouverture de connexion
  - Nécessaire avant tout transfert de données
  - Permet d'établir les numéros de séquence initiaux (ISN) et la taille des MSS
  - Flag SYN actif : ouverture de connexion
- Three Way handshake

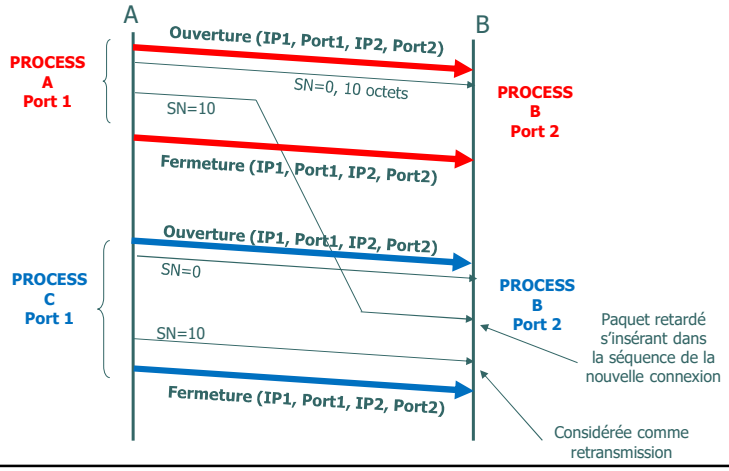


- Connexion en trois phases
  - Demande d'ouverture par A (SYN=1), choix de l'ISN_A=n
  - Acceptation par B (ACK n+1), choix de l'ISN_B =p
  - A acquitte l'acceptation (ACK p+1)

34

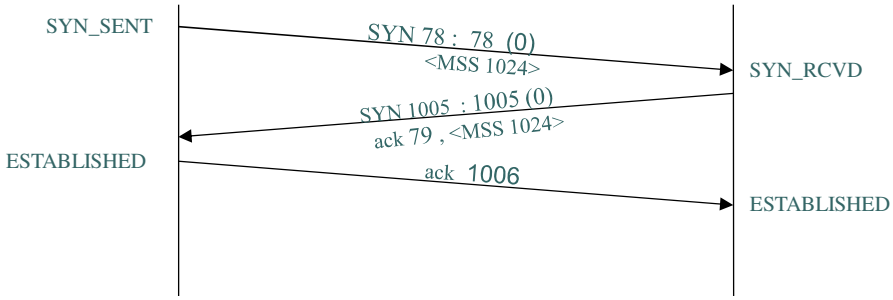
Connexions TCP – Ouverture de connexion

- ISN (Initial Sequence Number)
  - Permet de distinguer les octets de deux connexions successives utilisant les mêmes ports (éviter la confusion entre incarnations)
  - Choisi en se basant sur une horloge incrémentée toutes les 4 micro.sec)



35

Connexions TCP – Ouverture de connexion

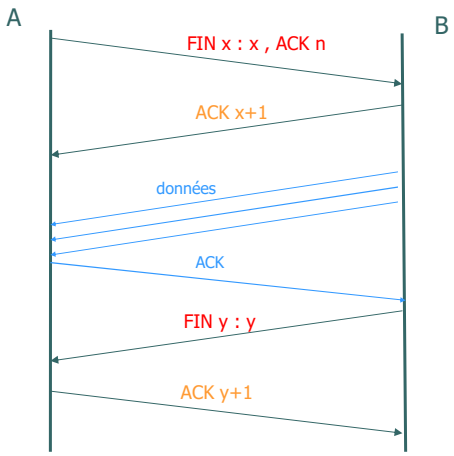


- MSS = Maximum Size
  - Spécifiée en fonction de la taille des tampons de réception
  - Si les 2 extrémités sont connectées au même réseau physique alors MSS = MTU

36

Connexions TCP – fermeture de connexion

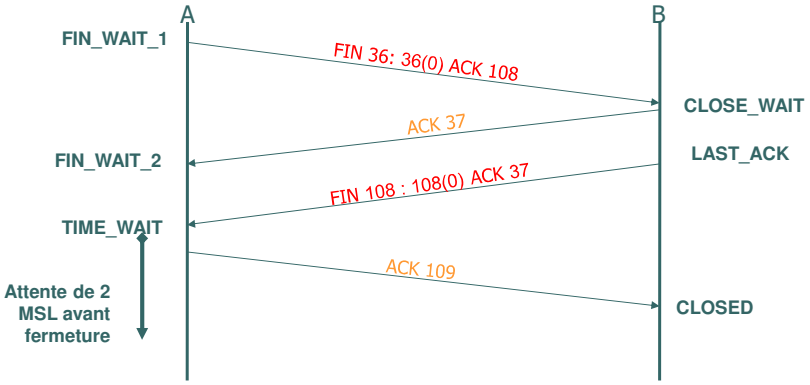
- Demande de fin de connexion (FIN) par A
- Acquittement du FIN (ACK) par B
- B envoie ses données en attente
- A acquitte les données (ACK)
- Acceptation de la fin de connexion de B (FIN)
- Acquittement de la fin de connexion (ACK) (plus de données à émettre)



37

Connexions TCP – fermeture de connexion

- Flag FIN actif : plus de données à émettre

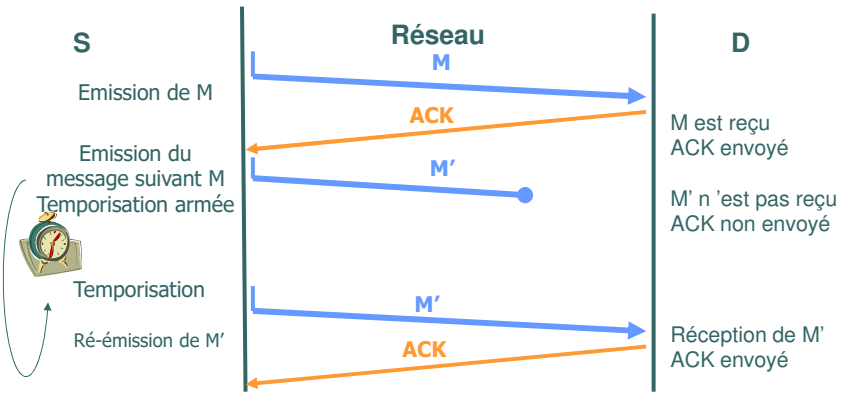


- Connexion TCP full-duplex
  - les données circulent indépendamment dans un sens et dans l'autre. Les deux directions doivent pouvoir être interrompues indépendamment l'une de l'autre.
- TIME_WAIT ou état d'attente 2MSL (Maximum Segment Life)
  - permet à TCP au niveau du client de retransmettre le dernier ACK si perdu (timeout du serveur et retransmission du dernier FIN)

38

Le protocole TCP - Fiabilité

- Principes : Acquittements-Temporisation-Retransmission
  - S émet un message M vers D. S attend un ACK de D avant d'émettre le message suivant M'
  - Si l'acquittement A_i ne parvient pas à S, S considère au bout d'un certain **temps** que le message est perdu et **reémet** M_i



39

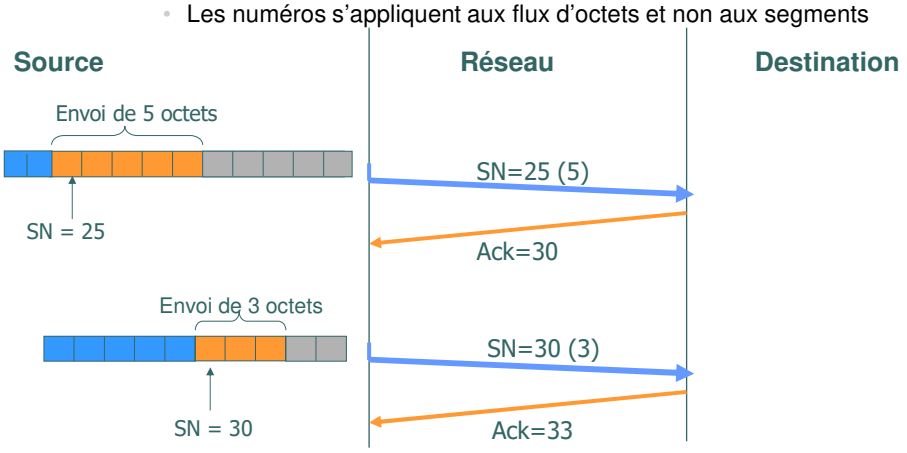
Le protocole TCP - Fiabilité

- Numérotation
  - Eviter la duplication des paquets (en cas de perte de l'acquittement)
- à chaque segment sont associés :
  - Numéro de séquence
    - Il indique le numéro du premier octet transmis dans le segment
  - Numéro d'acquittement
    - Il indique le numéro de séquence du prochain octet attendu
    - Acquittement cumulatif
      - tous les octets précédents sont implicitement acquittés
    - Numéro d'acquittement = Numéro de séquence du dernier segment reçu + Taille du segment reçu (données)

40

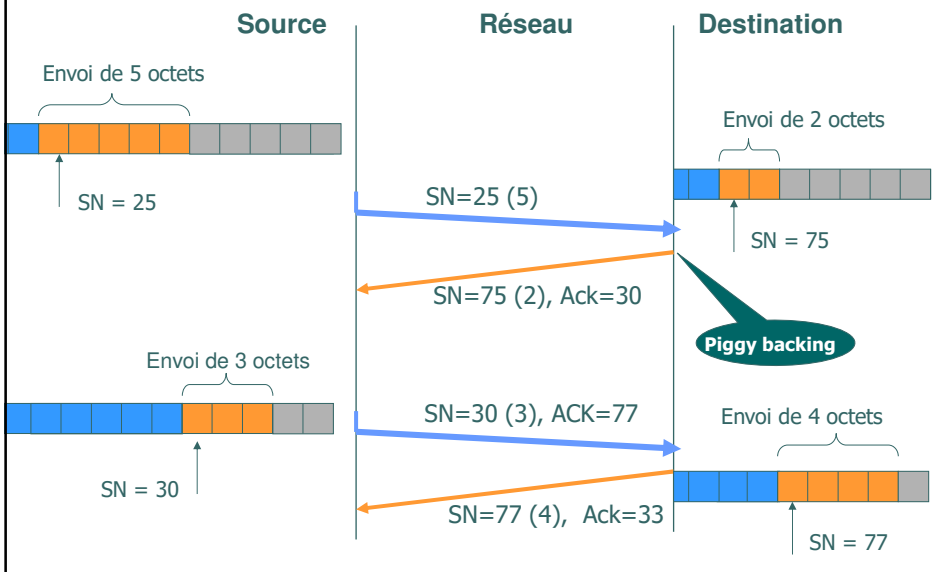
Le protocole TCP - Fiabilité

- Numérotation – Remarque
  - Pour TCP, les données à transférer se présentent sous forme de flux d'octets sans structure
  - Les numéros s'appliquent aux flux d'octets et non aux segments



41

Le protocole TCP - Fiabilité

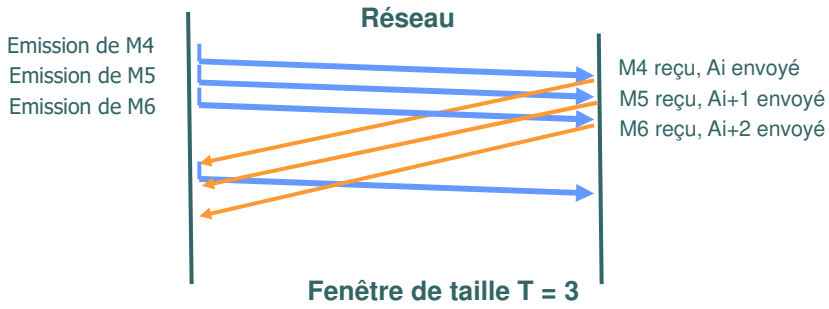


42

Le protocole TCP – Notion de fenêtre

- La technique d'acquittement simple pénalise les performances
- Le fenêtrage améliore le rendement du réseau (efficacité)

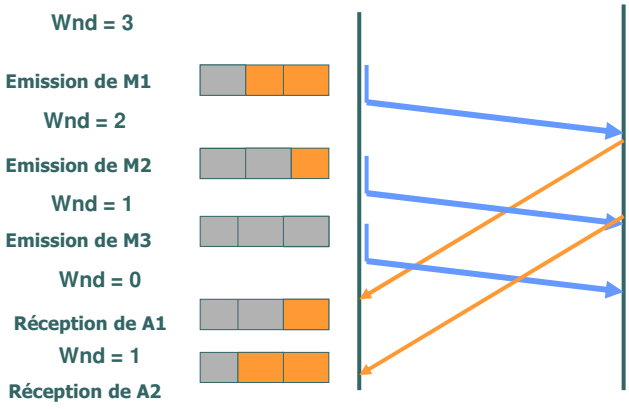
- Une fenêtre de taille T, permet l'émission d'au plus T messages "*non acquittés*" avant de ne plus pouvoir émettre



43

Le protocole TCP – Notion de fenêtre

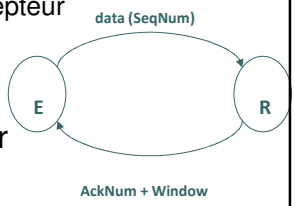
- La fenêtre se ferme à mesure que la source envoie des segments
- La fenêtre s'ouvre à mesure que les acquittements arrivent à la source



44

Le protocole TCP - Contrôle de flux

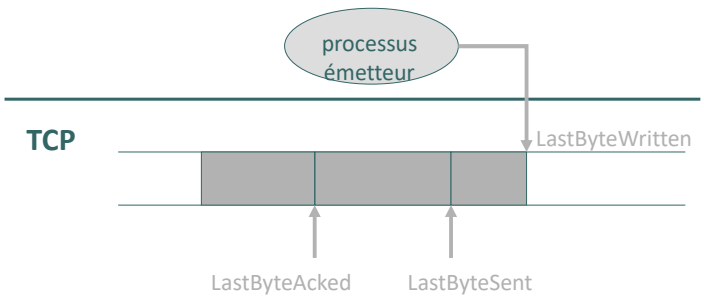
- Le fenêtrage impose de tenir compte de la capacité du récepteur, sinon perte de segments
- Contrôle de flux
  - Evite le débordement de la mémoire du récepteur
  - Ajuste le rythme d'envoi de l'émetteur
- Fenêtre de contrôle de flux
  - Fenêtre de transmission chez l'émetteur
  - Crédit d'émission
  - Taille contrôlée par le récepteur
    - champ *window size* (rwnd ou Advertised Window) (*Taille de fénêtre* dans le segment TCP)
    - Le récepteur indique dans ce champ le nombre d'octets qu'il peut effectivement recevoir
    - s'applique du numéro d'acquittement



45

Buffer d'émission

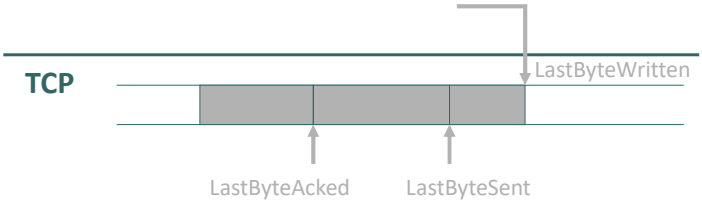
- en émission, un buffer stocke
  - les données envoyées et en attente d'acquittement
  - les données passées par le processus émetteur mais non encore émises



46

Buffer d'émission

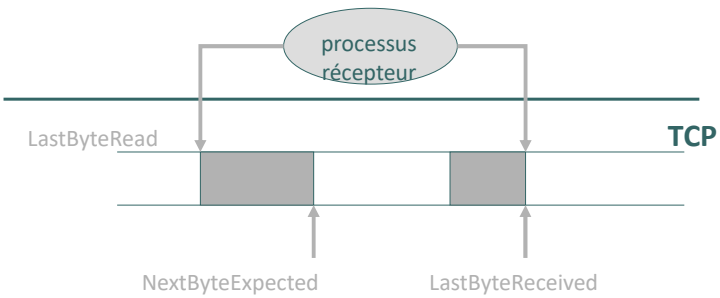
- Remarque : on a toujours
  - LastByteAcked <= LastByteSent <= LastByteWritten
- TCP vérifie à tout moment que
  - LastByteSent – LastByteAcked <= AdvertisedWindow
- TCP calcule une fenêtre effective
  - EffectiveWindow <- AdvertisedWindow – (LastByteSend – LastByteAcked)
- en parallèle, TCP doit s'assurer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



47

Buffer de réception

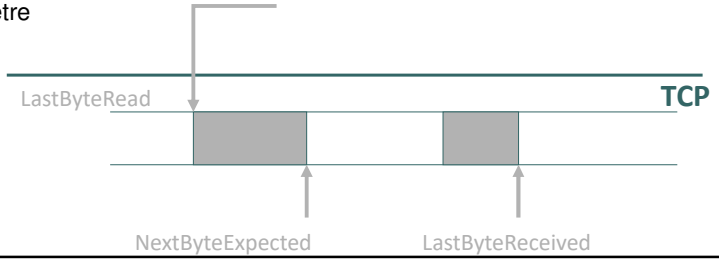
- en réception, un buffer stocke
  - les données dans l'ordre non encore lues par le processus récepteur
  - les données déséquencées



48

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



49

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

50

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

51

Dimensionnement du temporisateur

- algorithme initial
  - TCP calcule une estimation du RTT par une moyenne pondérée entre l'estimation précédemment calculée et le dernier échantillon mesuré du RTT
    - **EstimatedRTT <- α EstimatedRTT + (1 – α) SampleRTT**
    - SampleRTT est obtenu en mesurant le délai séparant l'émission d'un segment de la réception de son acquittement
    - 0,8 < α < 0,9
  - TCP calcule la valeur du temporisateur RTO (Retransmission Time Out)
    - RTO <- min (UBOUND, max (LBOUND, β 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 <= β <= 2

52

Dimensionnement du temporisateur

- Amélioration du calcul

$$MoyenneRTT = (1 - \alpha)\ MoyenneRTT + \alpha\ EchantillonRTT, \quad \alpha = 1/8, \beta = 1/4$$

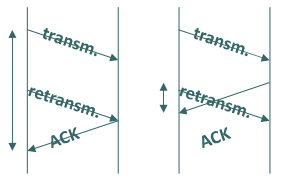
$$VarianceRTT = (1 - \beta) * VarianceRTT + \beta\ (|\ EchantillonRTT - MoyenneRTT\ |)$$

$$RTO = MoyenneRTT + 4\ VarianceRTT$$

53

Dimensionnement du temporisateur

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



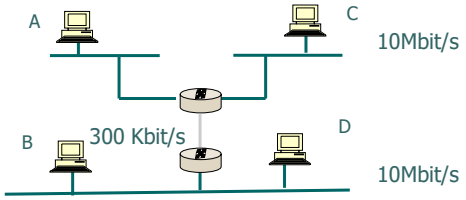
SampleRTT trop grand | SampleRTT trop petit

- 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

54

Le protocole TCP - Contrôle de congestion

- Contrôle de flux non suffisant pour éviter les pertes
- La saturation des nœuds (routeurs) peut entraîner des pertes
- Les retransmissions peuvent aggraver le phénomène de congestion



- Contrôle de congestion
  - La source doit s'adapter à la charge des routeurs intermédiaires
  - Le contrôle de congestion TCP ne requière aucune signalisation
  - Se base sur les manifestations indirectes de la congestion (retour des acquittements, pertes)
  - Réduire le débit de la source en cas de congestion et l'augmenter sinon (auto-adaptation)

55

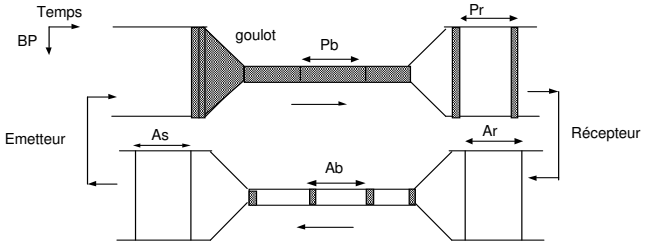
Effondrement du réseau

- Octobre 1986
  - Effondrement de l'Internet du à la congestion
  - Il y avait seulement contrôle de flux
    - pas de contrôle de congestion
- « Congestion avoidance and control »
  - Van Jacobson, *SIGCOMM* '88
  - A introduit*:*
    - self-clocking principle
    - AIMD & Slow Start
    - Fast Retransmit & Fast Recovery
    - improved RTT estimation

56

Contrôle de congestion TCP

- Conservation des paquets (*self clocking*)
  - Ne pas injecter un nouveau paquet tant qu'un vieux n'est pas sorti du réseau
    - nombre de paquets en transit constant
  - Synchronisation sur les acquittements: auto-synchronisation
    - les ACKs régulent (« clock ») l'émission des données



- Idée du contrôle de congestion TCP
  - Trouver le point d'équilibre
  - Emettre au rythme d'équilibre

57

Mise en oeuvre

- Maintenir une estimation de la capacité du réseau
  - Taille de fenêtre en situation d'équilibre: *ss_thres*
- Détermination du nombre d'octets maximum que peut contenir le réseau
  - Utilisation d'une fenêtre dynamique: fenêtre de contrôle de congestion (*cwnd*)

    wnd= min (rwnd, cwnd)
- Dynamicité de la taille de la fenêtre (AI-MD):
  - Augmentation additive
    - Sondage de la capacité disponible par croissance "additive"
    - Augmenter le taux d'utilisation des ressources en cas de sous utilisation
  - Diminution multiplicative en cas de congestion
    - Guérison de la congestion
    - Détermination du nouveau point d'équilibre
      - Ss_threshold = 1/2. nombre d'octets en transit
- Indication de congestion
  - Perte d'un segment
  - cwnd = 1 (ou autre selon version de TCP)
- Une estimation du RTT plus précise:
  - Prise en compte de la variance
- Emission à la réception d'un ACK
- Augmentation de la *cwnd* en 2 phases
  - phase 1: slow start
  - phase 2: congestion avoidance

58

Initialisation

- ss_thres = valeur arbitraire
- cwnd = 1 (dans la pratique valeur plus grande, ex. 10 pour linux depuis 2010)
- Slow start

59

Slow-Start

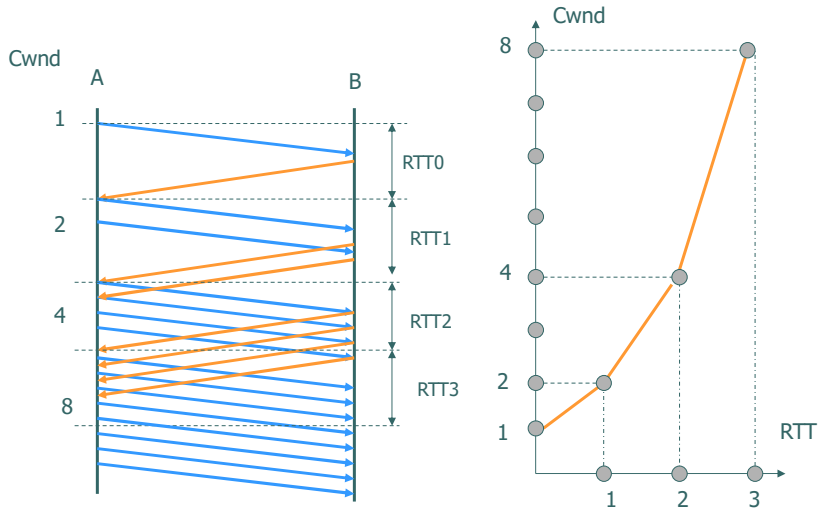
- Problème du démarrage
  - La transmission des données commence sans avoir connaissance de l'état du réseau
- Motivations du Slow Start
  - Eviter d'envoyer une rafale après une période d'inactivité
  - Atteindre rapidement le point d'équilibre
  - Acquisition rapide de l'auto-synchronisation
  - Détection rapide d'une sur-estimation du point d'équilibre
- Fonctionnement
  - cwnd =1
  - Incrémente cwnd de 1 segment par ACK (non dupliqué) reçu

    Cwnd = Cwnd + 1 (en nombre de segments)
  - Cela double la cwnd par RTT
  - Augmentation exponentielle de la fenêtre

60

Le protocole TCP - Contrôle de congestion

- RTT (Round Trip Time)



61

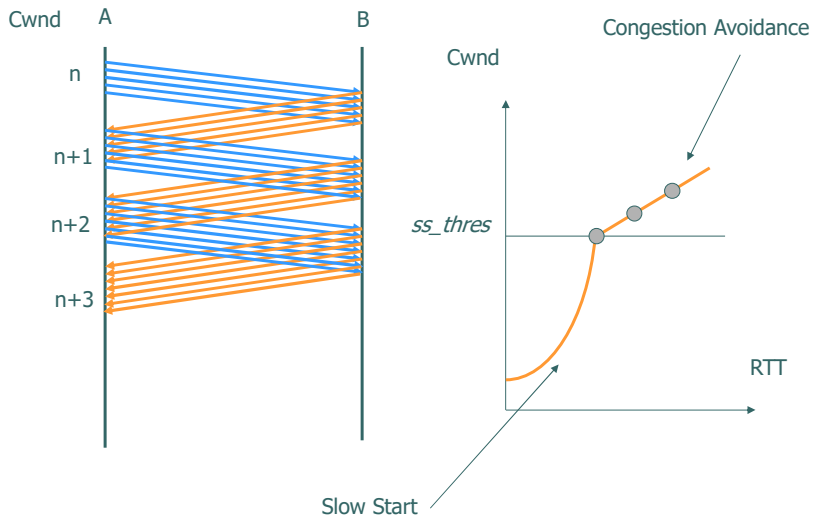
Le protocole TCP - Contrôle de congestion

- Motivations
  - Le point d'équilibre (ss_thres) est considéré atteint
  - Sonder le réseau doucement pour obtenir de la bande passante supplémentaire
- Congestion Avoidance
  - Lorsque Cwnd > ss_thres
  - A partir de ce seuil la croissance est linéaire (et non exponentielle)
  - Après le seuil,
    - l'incrémentation est d'un segment par RTT

      Ou
    - *Cwnd = Cwnd + 1/Cwnd à chaque Ack (non dupliqué) reçu*
  - Augmentation linéaire de la fenêtre

62

Le protocole TCP - Contrôle de congestion



63

Détection de pertes et réaction

- Détection de pertes
  - Deux cas
    - Cas 1 : Expiration de la temporisation
    - Cas 2 : Réception d'acquittements dupliqués

- Cas 1 : Echéance du temporisateur de retransmission

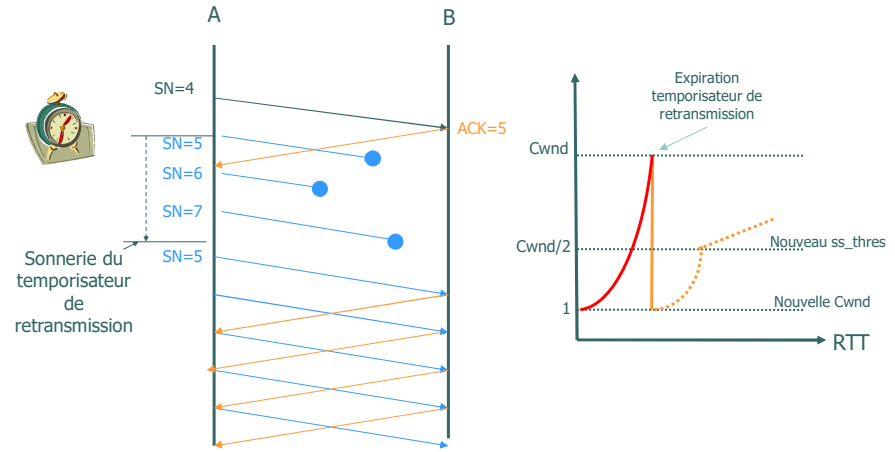
  Actions

  *ss_thres = Max(cwnd/2 , 2*MSS)*
  Cwnd = 1
  Passer à l'état Slow Start

64

Détection de pertes et réaction

Cas 1 : expiration du temporisateur

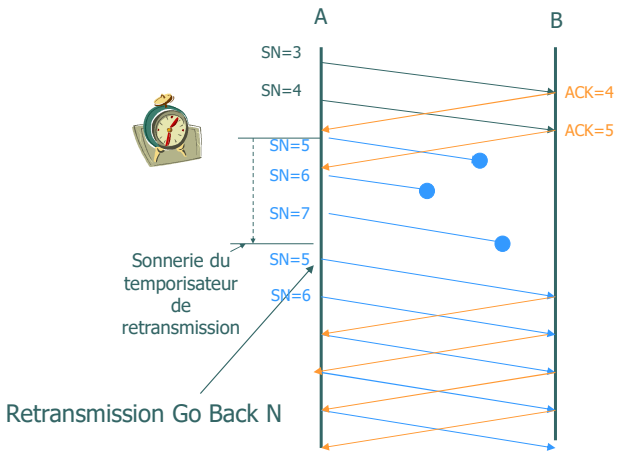


65

Détection de pertes et réaction

Cas 1 : expiration du temporisateur

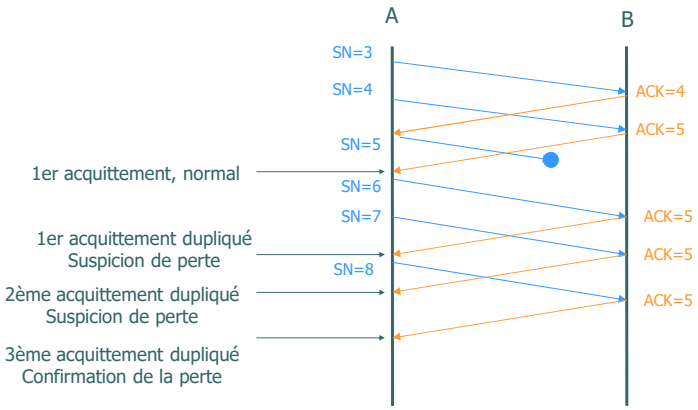
- Retransmission Go Back N
  - Retransmettre tous les segments à partir de la perte (celui dont le temporisateur a expiré)



66

Détection de pertes et réaction
Cas 2 : réception d'acquittements dupliqués

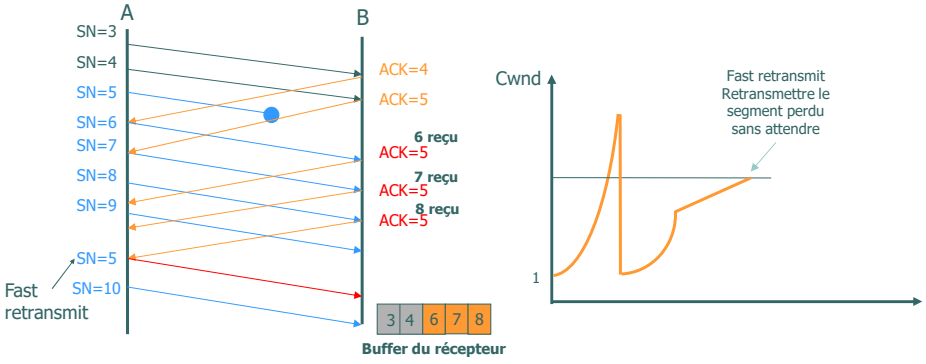
- Réception d'acquittements dupliqués
  - Si réception de 3 acquittements dupliqués (4 Ack identiques)



67

Détection de pertes et réaction
Cas 2 : réception d'acquittements dupliqués

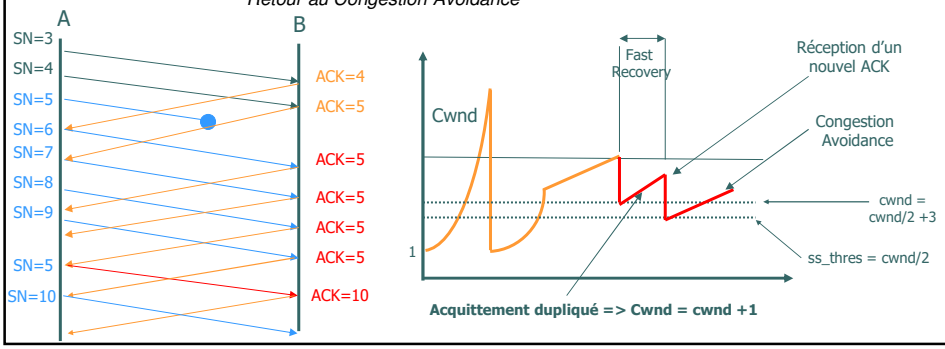
- Réception de 3 acquittements dupliqués
  - **Fast retransmit**
    - Retransmettre le message dont on suspecte la perte sans attendre le timeout (expiration du temporisateur)
    - Ne pas passer au Slow Start mais passer en état **Fast Recovery** (sauf Tahoe qui passe en slow start)



68

Détection de pertes et réaction
Cas 2 : réception d'acquittements dupliqués

- Réception de 3 acquittements dupliqués (suite)
  - **Fast recovery**
    - *ss_thres = Max (cwnd/2, 2*MSS)*
    - cwnd = *ss_thres* + 3 (« gonflement de cwnd », 3 car 3 segments déjà dans le buffer du récepteur)
      Envoi éventuel de nouveaux paquets
    - Pour chaque acquittement dupliqué reçu alors :
      cwnd = cwnd + 1
      Envoi éventuel de nouveaux paquets
    - Si réception d'un nouvel ACK (non dupliqué) alors :
      cwnd = ss_thres (« dégonflement de cwnd »)
      Retour au Congestion Avoidance



69

Le protocole TCP - Contrôle de congestion (CC)
Evolution des versions

- 1974–1980 Protocoles TCP et UDP, les deux sans CC
- Les versions avec CCs classiques :
  - 1988 TCP Tahoe, 1er CC, slow start + cong. avoidance + fast retrans.
  - 1990 TCP Reno, Tahoe + fast recovery, se remet plus rapidement lors d'une perte
  - 1994 TCP Vegas, basé sur l'historique du RTT (état des routeurs)
  - 1999 TCP NewReno, TCP Reno + adaptation de frec, gère mieux plusieurs pertes
- Options (entre l'émetteur et le récepteur) :
  - 1992 Window scaling et timestamps, gère de grandes fenêtres de réception, resp. mesure les RTTs
  - 1996 SACK, DSACK, gère mieux les pertes en spécifiant exactement les paquets reçus
- 1994 ECN, les routeurs donnent des informations sur la congestion
- 1999/2003 Algorithme ABC, modification d'un CC, des octets à la place de paquets
- 2000 TFRC, CC basé équation
- CCs réseaux sans fil :
  - 2001 TCP Westwood+, basé sur l'historique du RTT, meilleure utilisation si pertes aléatoires

70

Le protocole TCP - Contrôle de congestion (CC) Evolution des versions

- CCs pour grands cwnd (réseaux « rapides ») :
  - 2003 High-speed TCP, suivi de 2004 BIC, 2005 CUBIC, 2006 Compound TCP
- 2006 Protocole DCCP, entre UDP et TCP, sans retransmission, choix du CC
- 2013 Algorithme PRR, se remet plus rapidement lors d'une perte dans les flux courts (Web)
- 2013 Data Center TCP
- Beaucoup d'autres...

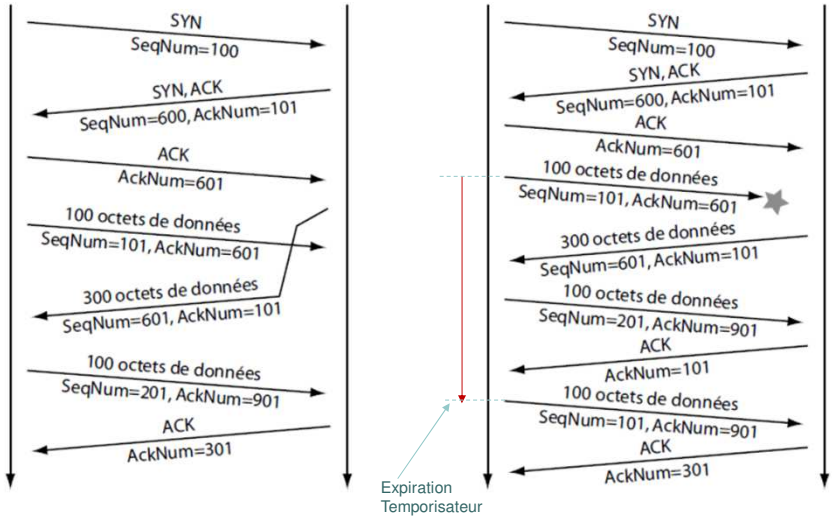
71

Exercice 1

- On considère une connexion TCP entre deux machines distantes A et B. A doit envoyer à B deux segments de 100 octets de données chacun et B doit envoyer à A un segment de 300 octets de données. On suppose que A est à l'origine de l'établissement et que l'ISN de A est égal à 100 et celui de B à 600.
- Représenter le diagramme de l'échange entre les deux machines
- Représenter l'échange si le 1er segment de A se perd

72

Exercice 1



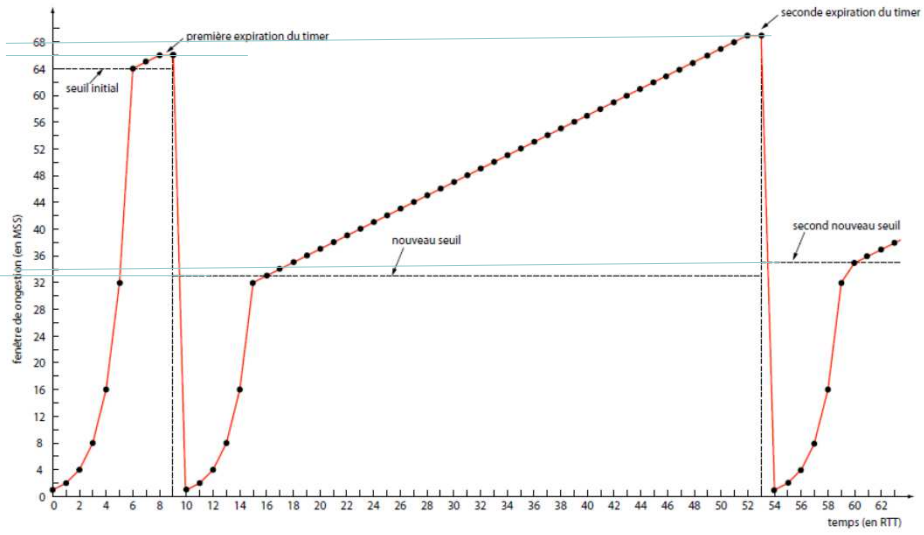
73

Exercice 2

- Tracez la courbe illustrant les variations de la taille de la fenêtre de congestion de TCP sous les hypothèses suivantes :
  - MSS = 1024 octets
  - Ss_thres initialisé à 64 Ko
  - Après 9 RTT, le temporisateur de retransmission expire
  - Le temporisateur de retransmission expire à nouveau après 44 RTT

74

Exercice 2



75

Exercice 3

- Soit un hôte A qui veut envoyer un gros fichier à un hôte B via une connexion TCP. Le délai RTT entre les deux hôtes est de 20ms via une liaison à 1Gbit/s. On suppose que le fichier est émis en utilisant des segments de 1000 octets. On suppose également que les ACK sont petits et peuvent être ignorés.
  - 1) quelle est la taille minimale de la fenêtre (en segments) pour que le taux d'utilization du lien soit supérieur à 80%?
  - 2) En supposant un seuil ss_threshold initial infini, aucune perte et aucun flux compétitif, combien de temps faudrait-il au mécanisme slow start pour atteindre les 80% de taux d'utilization?

76

Exercice 3 : corrigé

- RTT = 20ms
- Débit = 1Gbit/s
- Segment =1000 octets

1) quelle est la taille minimale de la fenêtre (en segments) pour que le taux d'utilization du lien soit supérieur) à 80%?

- Produit Bande X délai de A à B = $10^9 * 20 * 10^{-3} = 2 * 10^7$ bits
- Pour 80% de taux d'utilization il serait : $0,8 * 2 * 10^7$ bits= $1,6 * 10^7$ bits
- Taille de la fénêtre en segments : $1,6 * 10^7 / 8*1000 = 2000$ segments

2) Combien de temps faudrait-il au mécanisme slow start pour atteindre les 80% de taux d'utilization?

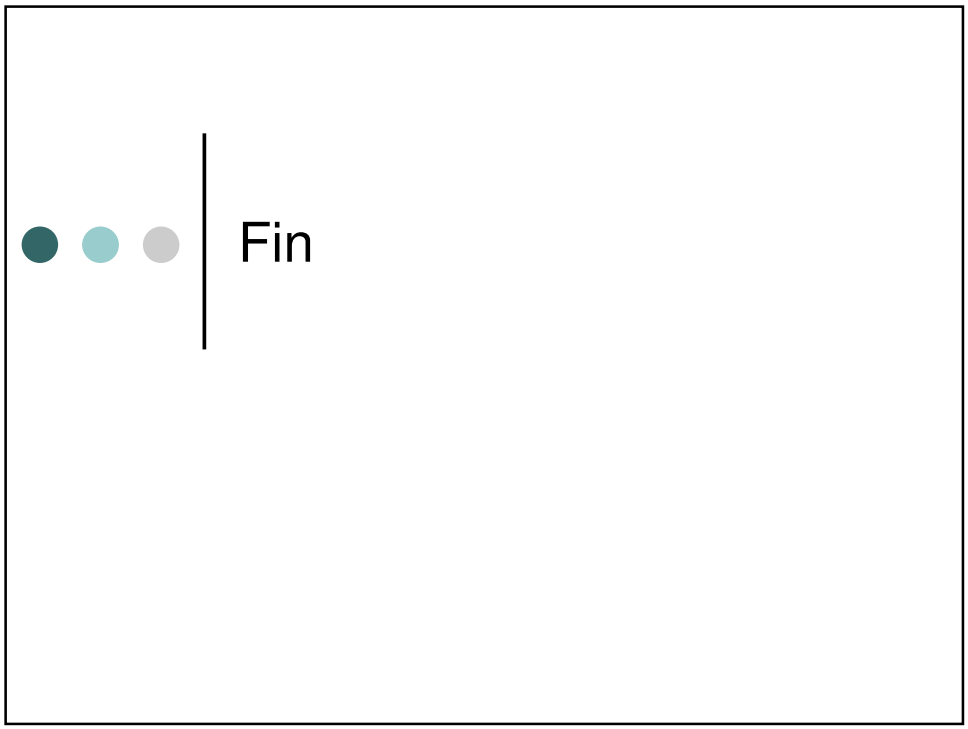
- Nombre de RTT = $\lceil \log_2(2000) \rceil = 11$
- Temps = $11 * 20 * 10^{-3} = 0,22$ sec

77

Versions de TCP

- Unix
  - BSD 4.2 (1983) 1ére souche TCP/IP largement disponible
  - BSD 4.3 Tahoe (1988), slow start, congestion avoidance, fast retransmit
  - BSD 4.3 Reno (1990), fast recovery
  - BSD 4.3 New Reno (1990), amélioration de Reno
  - BSD 4.3 Vegas (1990), Estimateur de BP

78



79