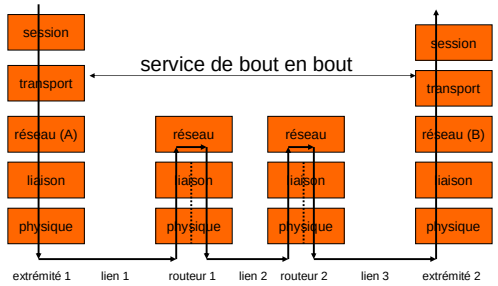
Chapitre 5 La couche transport

- 1. Service transport
  - 1.1. Service fourni aux couches supérieures
    - Assure l'acheminement de messages (TPDU) entre deux applications distantes avec certaines qualités (fiabilité, ordonnancement, délai, …)
    - « cache » les réseaux sous-jacents aux applications
    - Plusieurs services possibles
      - orienté connexion, fiable (ex. TCP)
      - sans connexion, pas fiable (ex. UDP)
    - négociation éventuelle d'options lors de l'établissement d'une connexion transport

1

La couche transport



2

1.2. Les primitives du service transport

- permettent aux processus d'application d'accéder au service transport
- services en mode connecté de type TCP :
  - point à point, analogie avec les « tubes » Unix
  - émission / réception d'un flot de données
  - le découpage des messages n'est pas préservé par le protocole transport
  - d'autres services (ex ISO) maintiennent le découpage
- services en mode non connecté de type UDP
  - émission / réception d'un message complet comportant l'adresse complète de destination
  - peut être point à multipoint (broadcast, multicast)

3

1.2. Les primitives du service transport

- cas d'un service en mode connecté
  - les primitives permettent au programme d'application d'établir, utiliser et libérer les connexions
  - en général mode dissymétrique client/serveur
- exemples de primitives
  - listen (pas de TPDU associé)
  - connect (=> TPDU Connection Request et Connection Accepted)
  - send (=> TPDU Data)
  - receive (pas de TPDU associé)
  - disconnect (=> TPDU Disconnection Request)
    - déconnexion asymétrique : fermeture à la demande d 'une des deux extrémités
    - déconnexion symétrique : fermeture dans les deux sens séparément

4

1.2. Les primitives du service transport

- Au moment de la connexion
  - identifier un point d'accès au service (TSAP)
    - en général adresse réseau + sélecteur
      - ex adresse IP + N° port : port 80 = service http
  - identifier une connexion

    TSAP + sélecteur

    multiples connexions vers le même service

    exemple TCP : TSAP source + TSAP destination

5

1.2. Les primitives du service transport

- Exemple des sockets STREAM (TCP) de Berkeley :
  - socket : création d'une socket
  - bind : attache une adresse locale à une socket (TSAP)
  - listen : le serveur alloue l'espace pour mettre en file d'attente les appels entrants
  - accept : bloque le serveur dans l'attente d'une connexion entrante => quand TPDU Connection Request arrive, l'entité transport :
    - crée une socket de service
    - éventuellement crée un processus associé à cette connexion et revient sur la socket d'écoute pour attendre de nouvelles connexions
  - connect : bloque le client dans l'attente du TPDU Connection Accepted
  - send / receive : envoi / réception de données
  - close : libération de connexion symétrique

6

2. Eléments de protocole transport

- problématique analogue au niveau liaison
  - fiabilité, contrôle de flux, …
- mais le service sous-jacent utilisé est un réseau et non un lien physique :
- adressage explicite
- établissement de connexion plus complexe
- problème d'errance des données dans le réseau
  - doublons
  - substitutions
  - les paquets peuvent arriver dans le désordre
  - les délais peuvent être plus grands et plus variables
  - (mémoires des routeurs intermédiaires)
- gestion du contrôle de flux et de congestion de bout en bout
  - plus difficile (vitesse de réaction, interaction entre plusieurs connexions)

7

2.1. Etablissement d'une connexion

- Adresser l'application distante
  - adresse de machine (ex IP) + adresse de point de connexion (exemple port TCP)
  - multiplexage/démultiplexage de connexions
  - Ex TCP : Ident. connexion =
  - @ source + @ destination + port source + port destination
- éviter les substitutions
  - confusion des données d'une ancienne connexion avec celles d'une nouvelle connexion de même identificateur
    - on associe à chaque connexion une référence dont le modulo est très grand (TCP :32 bits)
    - négociation de la référence lors de la connexion
    - échange tripartite
- être « sûr » que la connexion est établie
  - « problème des deux armées »

8

2.2. Libération d 'une connexion

- libération asymétrique
  - brutale => perte de données
- libération symétrique
  - chaque sens est libéré indépendamment de l 'autre
  - bien adapté si les deux processus savent quand libérer la connexion
  - sinon, le dernier qui émet sa demande de déconnexion n 'est jamais sûr que celle-ci arrive
  - analogie avec le problème des deux armées
  - échange tripartite
  - besoin de temporisateurs dans les entités transport

9

3. UDP (User Datagram Protocol)

- Décrit dans le RFC 768
- Protocole très simple (peu de fonctions)
  - sans connexion, non fiable
    - permet de transporter des données utilisateur entre des numéros de port source et destinataire
    - préserve les limites des messages
    - applications de type transactionnel ou requête/réponse
    - applications multipoint
  - format de l'en-tête UDP
    - ports source et destination : identifient les processus d'application correspondants
    - longueur UDP : longueur du datagramme complet
    - total de contrôle (facultatif) : inclut un « pseudo en-tête » extrait de l'en-tête IP
    - données

10

4. TCP (Transmission Control Protocol)

- Décrit dans les RFC 793, 1122, 1323, …
  - constante évolution
  - 4.1. Modèle de service TCP
    - orienté connexion, **bidirectionnelle**
    - **fiable** : gère les pertes, remet les données dans l'ordre
    - assure un **contrôle de flux** entre émetteur et récepteur
    - une connexion TCP est identifiée par deux extrémités (adresses des sockets émetteur et destinataire)
    - les données sont véhiculées sous forme de flots d'octets
      - pas de délimitation des messages de bout en bout
      - équivalent d'un tube Unix
    - il existe un service de données urgentes
    - assure (implicitement) **contrôle de congestion** d'Internet

11

4.2 TCP : principes

- un message TCP (**segment**) est transmis dans un paquet IP
- tout octet de données transmis sur une connexion TCP est référencé par un n° de séquence (32 bits)
- un message TCP est formé d'un en-tête d'au moins 20 octets suivi éventuellement d'options et de données
- la taille d'un segment est limitée par :
  - la charge utile d'IP (maximum 65 535 octets)
- le «Path MTU discovery » peut être mis en oeuvre par TCP pour éviter la fragmentation
- TCP utilise une fenêtre d'anticipation en émission
  - fenêtre de taille variable (contrôle de flux et congestion)
  - et éventuellement en réception (SACK)
- l'entité réceptrice acquitte avec le n° du prochain octet attendu (ACK cumulatif)
- RFC 1106 implémente la retransmission sélective

12

4.2.1 Format entête TCP

bits 0 31

Port Source	Port Destination
Numéro Séquence	
Numéro Acquittement	
Taille entête et flags	Fenêtre
Checksum	Pointeur Urgent
Options éventuelles + bourrage	
Données éventuelles (nb entier d'octets)	

13

4.2.1 Format entête TCP

- ports source et destinataire (16 bits) : identifient les extrémités locales de la connexion
- n° de séquence et n° d'acquittement (32 bits) : chaque n° est relatif à un octet de données
  - N° séquence = N° du premier octet du segment si non vide
  - (sinon prochain à envoyer)
  - N° acquittement = prochain octet attendu
- taille de l'en-tête TCP (4 bits) : nombre de mots de 32 bits de l'en-tête (5 minimum + options)
- Flags (bits indicateurs) : rôle et contenu du segment
  - URG : présence pointeur de données urgentes valide
  - ACK : champ accusé de réception valide (connexion/déconnexion)
  - PSH : « donnée poussée » si = 1 (force livraison)
  - RST : réinitialiser la connexion (refus de connexion)
  - SYN : synchroniser les n° de séquence (connexion)
  - FIN : libération d'une connexion
  - également indicateurs de congestions (ECN, RFC 2481)

14

4.2.1 Format entête TCP

- taille de fenêtre W (16 bits) : contrôle de flux explicite
  - si = 0 : blocage de l'émetteur
  - si > 0 : indique combien d'octets peuvent être transmis à partir de N° Acquittement
- total de contrôle (16 bits) qui porte sur
  - en-tête et données TCP
  - « pseudo en-tête » IP (car @IP identifient connexion)
- pointeur d'urgence (16 bits) : quand le bit URG est positionné, ce champ repère, dans la fenêtre, la position où les données urgentes se terminent
- option :
  - certaines options utilisées à la connexion (négociation des capacités)
  - d'autres utilisées en cours de connexion

15

Exemples d'options

- MSS (Maximum Segment Size) : taille maximale que l'entité TCP peut recevoir dans son tampon (minimum du MSS des deux extrémités).
  - le MSS est l'unité de mesure de la fenêtre de congestion
- Window scale
  - permet d'augmenter la taille de la fenêtre
- Timestamp : horodatage
- Possibilité de retransmission sélective (SACK accepted)
- etc

- Après les options, éventuel bourrage pour aligner sur un mot de 32 bits

16

4.2.2. Gestion des connexions TCP

- Ouverture de connexion
  - tripartite : garantit que les deux extrémités sont prêtes à transférer des données et se sont accordées sur leurs numéros de séquence (≠ dans chaque sens)
  - 1. le client demande une connexion (CONNECT)
    - => segment TCP avec bit SYN = 1 et ACK = 0
  - 2. quand ce segment arrive à destination, s'il existe une application à l'écoute sur le port destinataire (serveur ayant exécuté LISTEN et ACCEPT), elle peut :
    - accepter la connexion : => segment avec SYN = ACK = 1
    - sinon refuser la connexion : => segment avec RST = 1
  - 3. quand ce segment arrive à destination, le client sait que le serveur est connecté, et il informe le serveur qu'il est aussi connecté => segment avec SYN = ACK = 1
  - Segments avec SYN contiennent valeurs initiales pour N° SEQ
    - 1 de moins que le premier octet envoyé

17

4.2.2. Gestion des connexions TCP

- Fermeture de connexion
  - les deux sens sont libérés indépendamment (fermeture symétrique)
  - en principe il faut 4 segments pour fermer une connexion TCP : un FIN et un ACK pour chaque sens
  - le premier ACK et le second FIN peuvent être dans le même segment => 3 segments
  - pour éviter le problème des deux armées, on arme un temporisateur lorsque l'on envoie un segment FIN
- pour une fermeture « correcte »
  - fermeture de l'application au préalable

18

4.2.3 Transfert de données

- Transfert bidirectionnel simultané
- Transfert de données
  - la gestion des fenêtres d'anticipation est liée :
    - aux réceptions d'acquittements
    - au rythme de lecture / écriture des applications
- L'émetteur peut envoyer les octets compris entre
  - dernier N° ACK reçu
  - et
  - dernier N°ACK reçu + dernier W reçu
- W permet donc au récepteur de ralentir émetteur
  - W = « crédit émission »

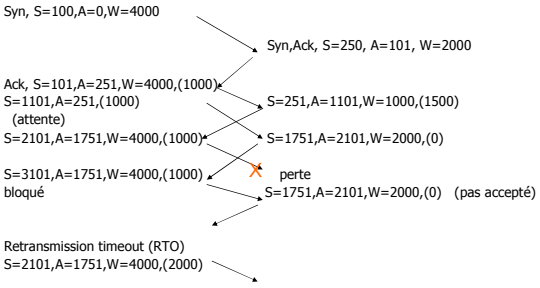
19

4.2.3 Transfert de données

- Fiabilité
  - récepteur accepte dans l'ordre et acquitte
  - émetteur arme délai de retransmission RTO
    - associé au plus ancien octet non acquitté
    - si RTO se déclenche : retransmission (continue)
  - Efficacité dépend estimation RTO
    - très variable suivant connexion
  - principe
    - RTT calculé pour chaque paquet acquitté : dernierRTT
    - NouveauRTT = a*AncienRTT + (1-a)*dernierRTT
    - RTO = b * NouveauRTT *(0 < a < 1, b > 1)*
    - algo de Karn : ne pas tenir compte des paquets retransmis

20

Exemple échange



21

4.2.3 Transfert de données

- quand W = 0, l'émetteur n'envoie plus de données sauf dans deux cas :
  - données urgentes
  - segment d'un octet : oblige le récepteur à ré-annoncer le prochain octet attendu et la taille de la fenêtre
    - => évite les interblocages si l'annonce d'une taille de fenêtre > 0 s'est perdue
  - les émetteurs ne sont pas tenus de transmettre immédiatement les données qui proviennent des applications, ni les récepteurs d'acquitter le plus rapidement possible
    - si application émettrice envoie octet par octet (ex. Telnet) et que le récepteur acquitte immédiatement => surcharge du réseau
    - => retarder les acquittements (récepteur)

22

4.2.3 Transfert de données

- => algorithme de Nagle (émetteur) : quand une application produit des données octet par octet, on envoie le premier octet et on accumule le reste dans un tampon en attendant l'acquittement
  - l'algo de Nagle est largement implémenté, sauf pour applis de type X-Windows (sinon mouvements de souris en rafales)
- syndrome de la fenêtre stupide : lorsque l'appli émetteur envoie les données par grands blocs, mais que l'appli récepteur lit octet par octet
  - => tampon récepteur presque plein, et récepteur envoie segments avec fenêtre = 1 à l'émetteur
  - Solution de Clark : empêche le récepteur d'envoyer segment avec fenêtre = 1
  - => récepteur doit attendre que son tampon soit suffisamment vide pour envoyer une taille de fenêtre (par exemple taille du MSS ou moitié du tampon vide)

23

4.2.3 Contrôle de flux

- Débit avec TCP standard
  - W < 64 Ko
  - au plus W par RTT
  - Ex :
    - RTT = 1ms (LAN)
      - 64Ko /ms => 64 Mo/s ~500 Mb/s
    - RTT = 640ms (satellite)
      - 64 Ko / 640 ms ~800 Kb/s
  - intérêt du paramètre Wscale (Window scale)
    - Wscale = n => multiplie W par 2^n

24

4.2.3 Contrôle de flux

- Emetteur bloqué dans 2 cas
  - W = 0 (récepteur saturé)
    - => attendre de recevoir segment avec W >0
  - La fenêtre d'émission est pleine
    - W octets envoyés et non encore acquittés
      - si perte d'un Ack : attendre Ack suivant (cumulatif)
      - si perte données : attendre RTO et retransmettre
      - si aucune erreur : attendre Ack (au plus RTT)
        - peut indiquer que W **trop petit** car la fenêtre s'est remplie avant la réception de l'ACK

25

4.2.3 Contrôle de congestion

- Dans Internet, contrôle distribué dans les stations émettrices
  - TCP s'en charge, en diminuant le débit des données émises
    - =>TCP fait varier dynamiquement la taille de la fenêtre d'émission
  - perte (et retransmission)
    - interprété par TCP comme **congestion** dans le réseau
    - => TCP surveille les temporisateurs de retransmission pour détecter une congestion
  - TCP gère deux fenêtres :
    - « fenêtre de contrôle de flux » : relative à la capacité du récepteur (paramètre W)
    - « **fenêtre de congestion** » : relative à la capacité du réseau (paramètre CW)
    - => le nombre d'octets qui peut être envoyé doit être le minimum de ces fenêtres
    - FE = min (W, CW)

26

4.2.3 Contrôle de congestion

- TCP basique (Tahoe, années 80)
- Débit initial : doit être faible (capacité réseau inconnue)
- La fenêtre de congestion CW est mesurée en MSS
- algorithme du « démarrage lent » (Jacobson 88)
  - à l'établissement de la connexion, CW = 1
  - émetteur envoie un segment de taille maximum
  - à chaque acquittement sans timeout, CW = CW +1
  - => à chaque RTT sans perte CW double (croissance exponentielle)
  - CW est limité à W
  - => si W faible CW n'augmente plus
- Lors d'une retransmission (Timeout)
  - considère qu'il s'agit d'une congestion
  - redémarre en slow start avec CW = 1

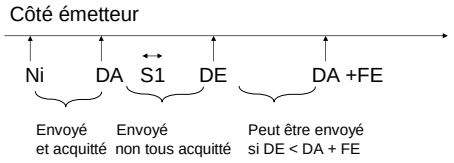
27

Retransmission sélective

- Nouveau mécanisme introduit RFC2018
  - négocié à la connexion
    - option SACK_permitted
  - si récepteur détecte une perte
    - N° Seq segment reçu > N° Seq segment attendu
    - récepteur envoie Ack avec
      - option SACK : contient n fois (début, fin)
      - indique réception de n blocs « isolés »
    - évite retransmission continue
    - permet de déclencher le fast retransmit/recovery

28

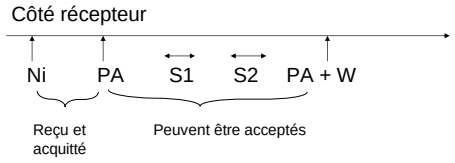
Fenêtres : synthèse



Ni : Numéro initial choisi par l'émetteur
DA : dernier N°Ack reçu
S1 : segment acquitté par SACK
DE : dernier octet envoyé
FE: fenêtre émission, min(W, CW)
A chaque acquittement
DA avance
FE est recalculé

29

Fenêtres : synthèse



Ni : Numéro initial choisi par l'émetteur
PA : Prochain attendu = dernier N°Ack envoyé
W : Fenêtre réception (fonction des capacités)
S1, S2 blocs reçus dans le désordre (SACK éventuel)
non encore fournis à l'application
A chaque segment attendu reçu
PA avance
W est éventuellement recalculé
Acquittement éventuellement envoyé

30

TCP : conclusion

- Protocole stable
  - amélioration algo retransmission/congestion
- Optimisation pour réseaux très haut débit
  - Wscale, SACK
- Manque d'efficacité si réseau peu fiable
  - perte ≠ congestion
  - Exemple : réseaux sans fil

31