Applications réseau

Cours 3 : Couche transport et programmation Java

Florian Sikora
florian.sikora@dauphine.fr

LAMSADE

M1 apprentissage

Adapté des slides de Kurose & Ross et d'E. Duris

Cours 3 : Couche transport

Principes de la couche transport

Multiplexage et démultiplexage

Transport sans connexion : UDP

Transport avec connexion : TCP

Objectifs

- (Rappels Licence) Les principes derrière les services de la couche transport.
  - Multiplexage / démultiplexage.
  - Transfert de données fiable.
  - Contrôle de flux.
  - Contrôle de congestion.
- (Rappels) Focus sur les services transports d'Internet
  - UDP, TCP.
- Programmation (Java) UDP/TCP pour coder des applications réseau.

Foire aux pseudos

http://www.quizzoodle.com/session/
445b23ec911040dfa8439e1cb7865306

Cours 3 : Couche transport

Principes de la couche transport

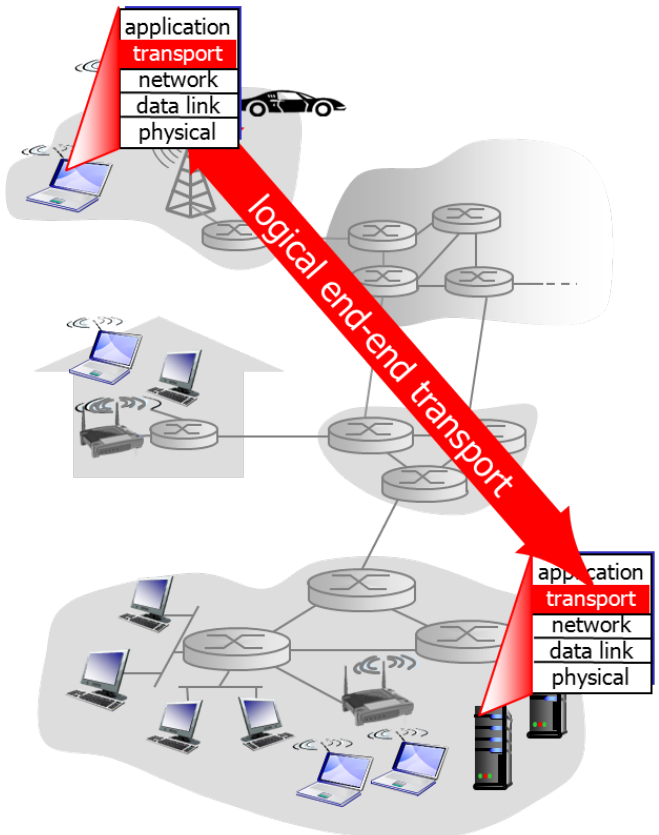
Multiplexage et démultiplexage

Transport sans connexion : UDP

Transport avec connexion : TCP

Services transport

- Fournir une communication logique entre des applications (processus) sur des hôtes différents.
  - Logique $\sim$ connexion directe pour les processus (selon leur point de vue), quelque soit la distance.
- Tournent sur les hôtes :
  - Partie envoi : casser le message de l'application en segments et le donner à la couche réseau.
  - Partie receveur : assembler les segments en messages et les donner à la couche application.
- Plusieurs protocoles disponibles pour les applications :
  - Internet : TCP et UDP.



Différences couche transport/réseau

- Couche réseau (L3) :
  - Communication logique entre hôtes, entre serveurs...
- Couche transport :
  - Communication logique **entre processus** (ne se préoccupe pas des caractéristiques physiques utilisées pour véhiculer les données).
  - Se repose sur les services de la couche réseau.

Différences couche transport/réseau

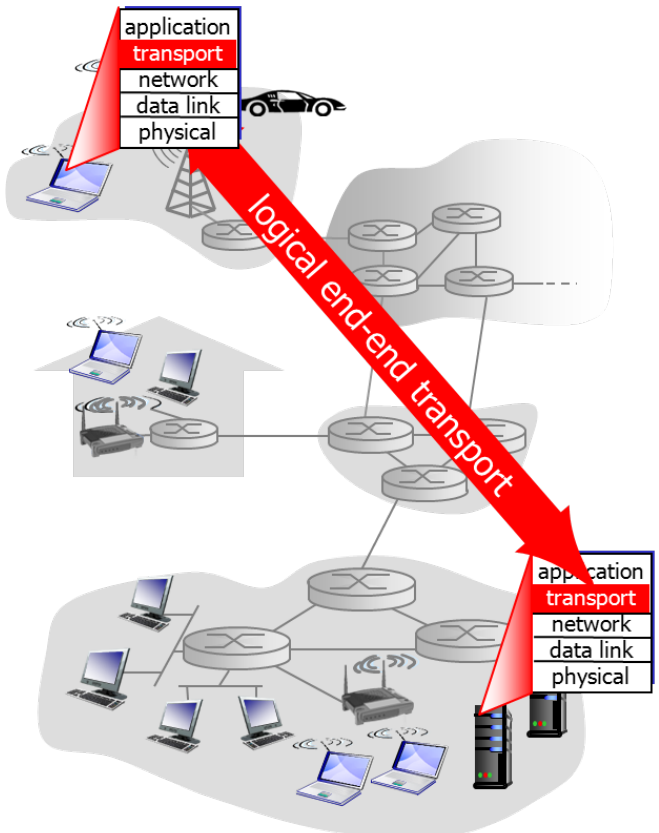
- (Tentative d') Analogie...
- 12 enfants dans la maison d'Anaïs qui envoient des courriers (vieille pratique des siècles précédents...) aux 12 enfants de la maison de Barack, leurs correspondants.
- Anaïs et Barack font la collecte du courrier sortant (et les donnent au facteur) et distribuent le courrier entrant.
  - Hôtes, terminaux
  - Processus
  - Messages d'application
  - Protocole de transport
  - Protocole couche réseau

Différences couche transport/réseau

- (Tentative d') Analogie...
- 12 enfants dans la maison d'Anaïs qui envoient des courriers (vieille pratique des siècles précédents...) aux 12 enfants de la maison de Barack, leurs correspondants.
- Anaïs et Barack font la collecte du courrier sortant (et les donnent au facteur) et distribuent le courrier entrant.
  - Hôtes, terminaux $\sim$ maisons, foyers.
  - Processus $\sim$ enfants, correspondants.
  - Messages d'application $\sim$ lettres dans les enveloppes.
  - Protocole de transport $\sim$ Anaïs et Barack (laison logique entre deux correspondants). Restent au foyer, ne sont pas concernés par la manière dont les courriers sont traités dans les centres de tris...
  - Protocole couche réseau $\sim$ La poste (liaison logique entre deux foyers).

Couche transport d'Internet

- Fiable, livraison dans l'ordre (TCP).
  - Contrôle des congestions.
  - Contrôle de flux.
  - **Mode connexion.**
- Non fiable, livraison sans ordre (UDP).
  - "Best effort", comme IP.
- Services non garantis :
  - Délais.
  - Bande passante allouée à l'application.



Cours 3 : Couche transport

Principes de la couche transport

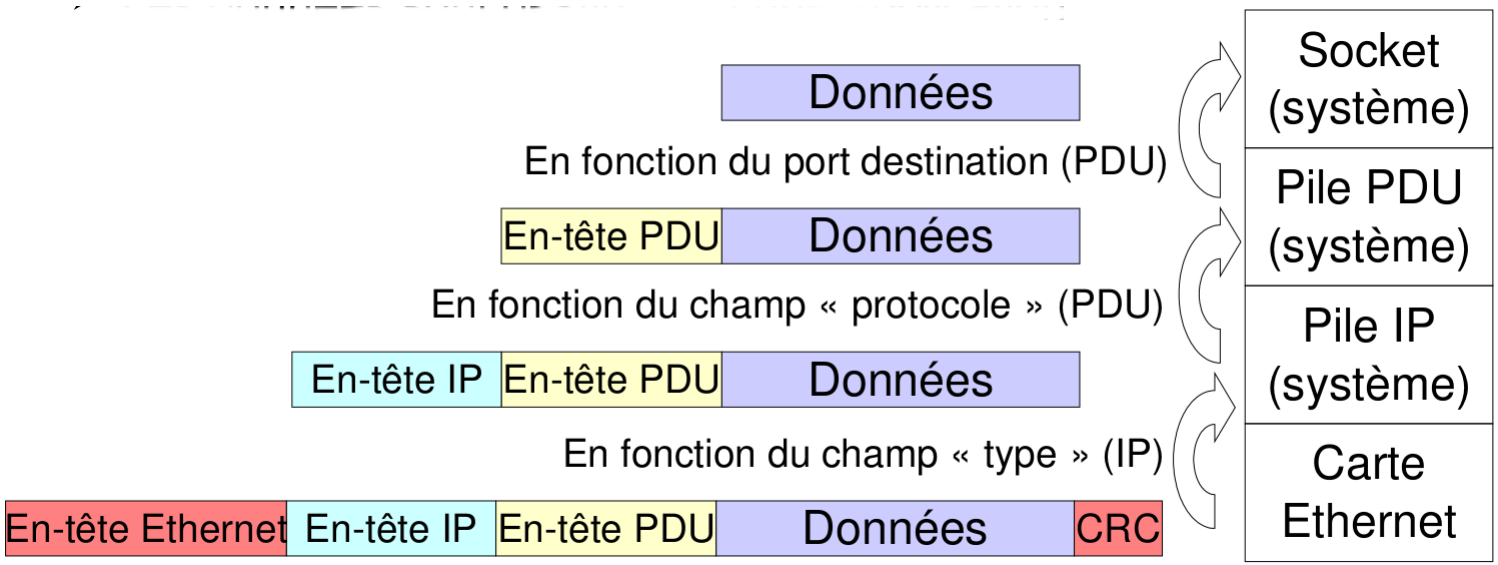
Multiplexage et démultiplexage

Transport sans connexion : UDP

Transport avec connexion : TCP

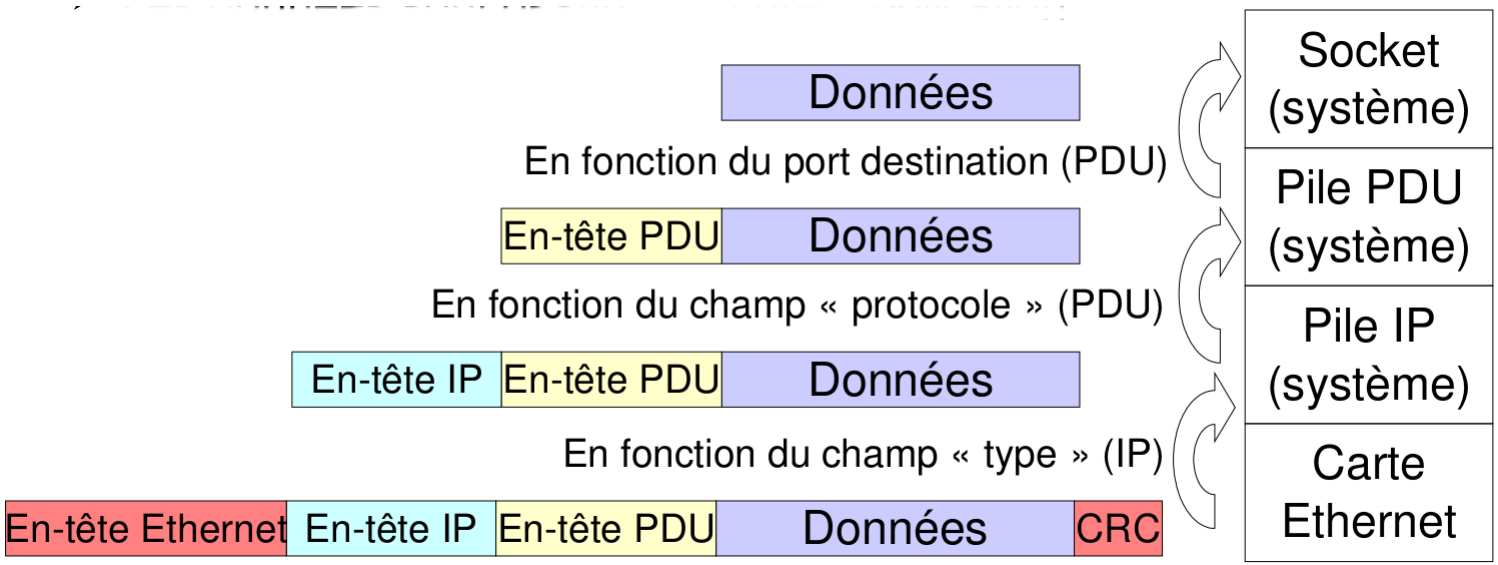
Encapsulation des données

- Comprendre comment le service expédition serveur à serveur de la couche réseau se transforme en service d'expédition processus à processus pour les applications.



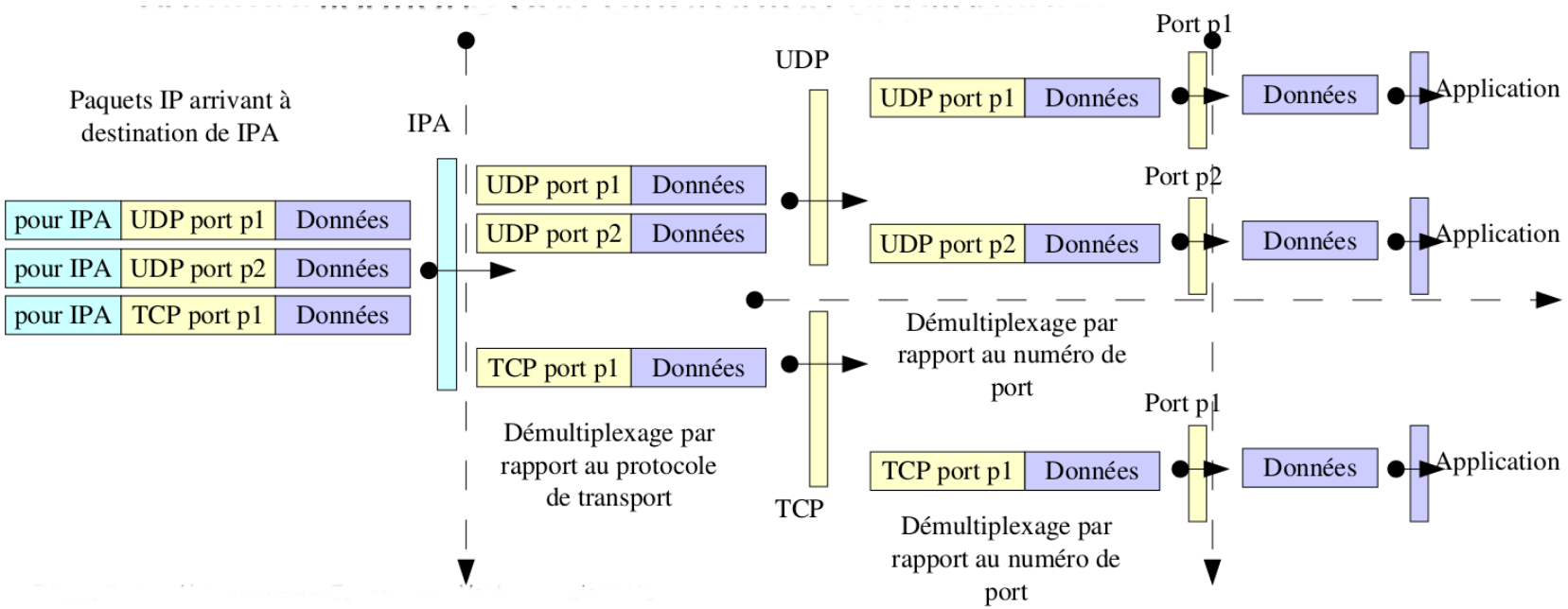
Encapsulation des données

- Comprendre comment le service expédition serveur à serveur de la couche réseau se transforme en service d'expédition processus à processus pour les applications.
- Trame Ethernet est adressée à une adresse MAC (couche 2).
- Paquet IP destinée à une adresse IP(couche 3).
- Un PDU (Protocol Data Unit) UDP ou TCP destiné à un port (couche 4).
- Données destinées à l'application.



Multiplexage et démultiplexage

- (Rappel) Applications liées à une socket pour l'accès au réseau.
- (Rappel) Socket identifiée par une adresse IP et un numéro de port.
- Socket permettent de multiplexer/démultiplexer les communications des différentes applis.

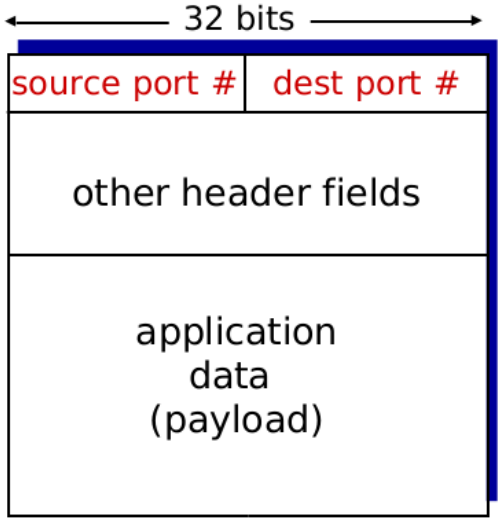


Multiplexage et démultiplexage

- Multiplexage : expéditeur. Prise en charge de plusieurs sockets, ajoute une en-tête (utilisée pour le démultiplexage).
- Démultiplexage : receveur. Utilise l'en-tête pour distribuer les segments reçus à la socket correcte.

Multiplexage et démultiplexage

- Nécessité :
  - Interface de connexion (socket) avec un identifiant unique.
  - Chaque segment porte un champ spécial indiquant la socket de destination.



TCP/UDP segment format

- Numéro de port : entre 0 et 65 535.
- Entre 0 et 1023 : “réservés”. (RFC 1700)

Multiplexage et démultiplexage

- Dans l'analogie précédente.
- Barack réceptionne le courrier depuis le facteur, et fait du démultiplexage en vérifiant à qui sont destinées les lettres et en les remettant à ses enfants.
- Anaïs fait du multiplexage en rassemblant les lettres et en les remettant au facteur.

Démultiplexage (sans connexion (UDP))

- Hôte reçoit un segment UDP :
  - Regarde le numéro de port destination.
  - Envoie du segment vers la socket ayant ce numéro de port.

Démultiplexage (sans connexion (UDP))

- Hôte reçoit un segment UDP :
  - Regarde le numéro de port destination.
  - Envoie du segment vers la socket ayant ce numéro de port.
- Segments avec le même numéro de port mais une source différente (IP ou port) sont dirigés vers la même socket.

Démultiplexage (avec connexion (TCP))

- Socket TCP identifiée par 4 éléments :
  1. IP de la source.
  2. Port de la source.
  3. IP destination.
  4. Port destination.
- Utilisation de ces 4 éléments pour diriger un segment vers la bonne socket.
  - Un serveur peut supporter plusieurs sockets TCP simultanément, chacune avec une identification différente.
  - Un serveur Web ouvre une socket pour chaque client.

Socket java

- Classe dédiée pour représenter une socket.
  - `java.net.DatagramSocket` pour UDP.
  - `java.net.Socket` pour TCP.
- Adresses sockets qui les identifient, indépendamment du protocole.
  - `java.net.SocketAddress` (classe abstraite)
  - En théorie indépendante du protocole réseau, mais en pratique une seule classe concrète (dédiée à IP) :
    - `java.net.InetSocketAddress`.

Socket java - InetSocketAddress

- `java.net.InetSocketAddress`, représente :
  - Une adresse IP (`InetAddress`)
  - Un port (`int`)

Socket java - InetSocketAddress

- `java.net.InetSocketAddress`, représente :
  - Une adresse IP (`InetAddress`)
  - Un port (`int`)
- Trois constructeurs :
  - `InetSocketAddress(InetAddress addr, int port)`
    - Si addr null, socket liée à l'adresse wildcard (n'importe quelle adresse IP locale, 0.0.0.0).
  - `InetSocketAddress(int port)`
    - IP wildcard.
  - `InetSocketAddress(String hostName, int port)`
    - Si hostName non résulu, adresse de socket marquée comme non résolue.
    - `isUnresolved()` pour tester.

Socket java - InetSocketAddress

- `IllegalArgumentException` si port en dehors de [0 ... 65535].
- Si port 0 : port éphémère choisi.
- `toString()` affiche “ip :port”.

```
1 InetSocketAddress sa1=new InetSocketAddress("truc.zarb",50) ;
2 System.out.println(sa1+(sa1.isUnresolved() ?"non res" :"res")) ;
3 // Affiche  : truc.zarb :50 non res
4 InetSocketAddress sa2=new InetSocketAddress("java.sun.com",80) ;
5 System.out.println(sa2+(sa2.isUnresolved() ?"non res" :" res")) ;
6 // Affiche  : java.sun.com/192.18.97.71 :80 res
```

Cours 3 : Couche transport

Principes de la couche transport

Multiplexage et démultiplexage

Transport sans connexion : UDP

Transport avec connexion : TCP

UDP User Datagram Protocol (RFC 768)

- C'est 2 paquets UDP qui discutent :
- "Ah bon ?"
- "Il parait que je peux arriver avant toi."

(DansTonChat.com)

UDP User Datagram Protocol (RFC 768)

- C'est 2 paquets UDP qui discutent :
- "Ah bon ?"
- "Il parait que je peux arriver avant toi."

(DansTonChat.com)

- Le problème avec UDP c'est que

(DansTonChat.com)

UDP User Datagram Protocol (RFC 768)

- Sans fioritures,
- “best effort”, les segments peuvent être perdus, délivrés dans le désordre à l’application.
- Sans connexion :
  - Pas de “handshaking” entre envoyeur et receveur.
  - Chaque segment géré de manière indépendante des autres.
- Q : Pourquoi UDP ?

UDP User Datagram Protocol (RFC 768)

- Sans fioritures,
- “best effort”, les segments peuvent être perdus, délivrés dans le désordre à l’application.
- Sans connexion :
  - Pas de “handshaking” entre envoyeur et receveur.
  - Chaque segment géré de manière indépendante des autres.
- Q : Pourquoi UDP ?
  - Pas de délai de connexion.
  - Simple, pas d’état de connexion d’un côté ou de l’autre.
  - Petite taille d’en-tête.
  - Pas de contrôle de connexion, peut aller vite !

UDP

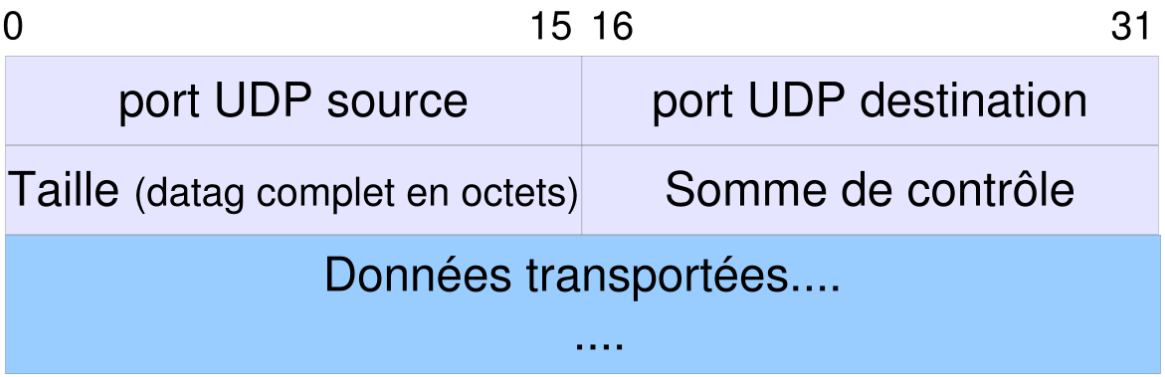
- Utilisé :

UDP

- Utilisé :
  - Streaming multimedia.
  - DNS.
  - SNMP (administrateur peuvent gérer du matériel à distance).
- Pour avoir de la fiabilité :
  - L'ajouter au niveau de la couche appli !

UDP

- Taille max des données transportées : $(2^{64} - 1 - 8) \sim 64Ko$.
- Checksum optionnel IPv4, obligatoire IPv6. Inclus IP dest et src.



- Exemple d'encapsulation dans une trame ethernet.

UDP checksum

- But : détecter des “erreurs” (par ex. bits inversés) dans le segment transmis.
  - Émetteur :
    - Traite le contenu du segment, en-tête inclus, comme une séquence d'entier sur 16 bits.
    - Checksum : addition (en complément à 1) des entiers (les retenues dues aux bits les plus significatifs sont ajoutées aux bits les moins significatifs).
    - L'ajoute dans le champ.
  - Receveur :
    - Calcule le checksum de ce qui est reçu.
    - Egal au champ checksum ?
    - NON : une erreur détecté, paquet supprimé. Pas de correction.
    - OUI : Pas d'erreur détecté (mais il y en a peut être quand même...).

UDP checksum - exemple

- Trois mots de 16 bits à sommer.

  0110011001100110

  0101010101010101

  Donne :

UDP checksum - exemple

- Trois mots de 16 bits à sommer.

0110011001100110

0101010101010101

Donne :

1011101110111011

Ajout du 3ème :

1011101110111011

0000111100001111

Donne :

UDP checksum - exemple

- Trois mots de 16 bits à sommer.

0110011001100110

0101010101010101

Donne :

1011101110111011

Ajout du 3ème :

1011101110111011

0000111100001111

Donne :

1100101011001010

Complément à 1 (checksum final) :

0011010100110101

Destinataire fait toute la somme (des mots + le CS) : doit donner

1111111111111111

UDP avec Java

- Version “classique” (avant 1.4), deux classes pour manipuler des tableaux d'octets.
- Tableaux à interpréter en fonction de l'application (encodage, format...).

UDP avec Java

- Version "classique" (avant 1.4), deux classes pour manipuler des tableaux d'octets.
- Tableaux à interpréter en fonction de l'application (encodage, format...).

- `java.net.DatagramSocket`.
  - Représente une socket d'attachement à un port UDP.
  - Une sur chaque machine pour communiquer.
- `java.net.DatagramPacket`. Représente 2 choses :
  - Les données qui transitent :
    - Tableau d'octets.
    - Indice de début.
    - Nombre d'octets.
  - Machine distante :
    - Adresse IP.
    - Port.

UDP avec Java - datagramSocket

- Objet pour envoyer ou recevoir des datagrammes UDP.
- Plusieurs constructeurs, avec différents arguments :
  - `InetSocketAddress`
    - Si null, socket non attachée, à attacher avec `bind()`.
  - port + `inetAddress`
  - port seul.
    - Sur adresse wildcard.
  - rien.
    - Sur adresse wildcard et un port libre (à éviter pour un serveur...).
- `SocketException` si problème d'attachement.

UDP avec Java - DatagramSocket

- `getLocalPort()`,`getLocalAddress()`,`getLocalSocketAddress(`
  - Donnent des infos sur l'attachement local de la socket.
- `bind(SocketAddress)`
  - Attache la socket à l'adresse si elle ne l'est pas déjà.
- `close()`
  - Ferme et libère les ressources.

UDP avec Java - DatagramPacket

- Représente un objet spécifiant les données à transporter ainsi que l'interlocuteur.
- Plusieurs constructeurs, spécifiant :
  - Les données, en octets bruts.
    - `byte[] buffer, int offset, int length`.
  - l'interlocuteur distant.
    - Soit `InetSocketAddress`.
    - Soit une `InetAddress` et un port (int).

UDP avec Java - DatagramPacket

- En émission, le DatagramPacket spécifie :
  - Les données à envoyer.
  - La machine et le port vers qui envoyer.
- En réception, spécifie :
  - La zone de réception des données.
  - La machine et le port depuis lesquels ces données ont été reçues.

UDP avec Java - DatagramPacket

- En émission, le DatagramPacket spécifie :
  - Les données à envoyer.
  - La machine et le port vers qui envoyer.
- En réception, spécifie :
  - La zone de réception des données.
  - La machine et le port depuis lesquels ces données ont été reçues.
- Un même objet peut servir aux deux usages.

UDP avec Java - émission et réception

- Un `DatagramPacket` est fourni et pris en charge par un `DatagramSocket`.
- Pour émettre : `dSocket.send(dPacket) ;`.
- Pour recevoir : `dSocket.receive(dPacket) ;`.

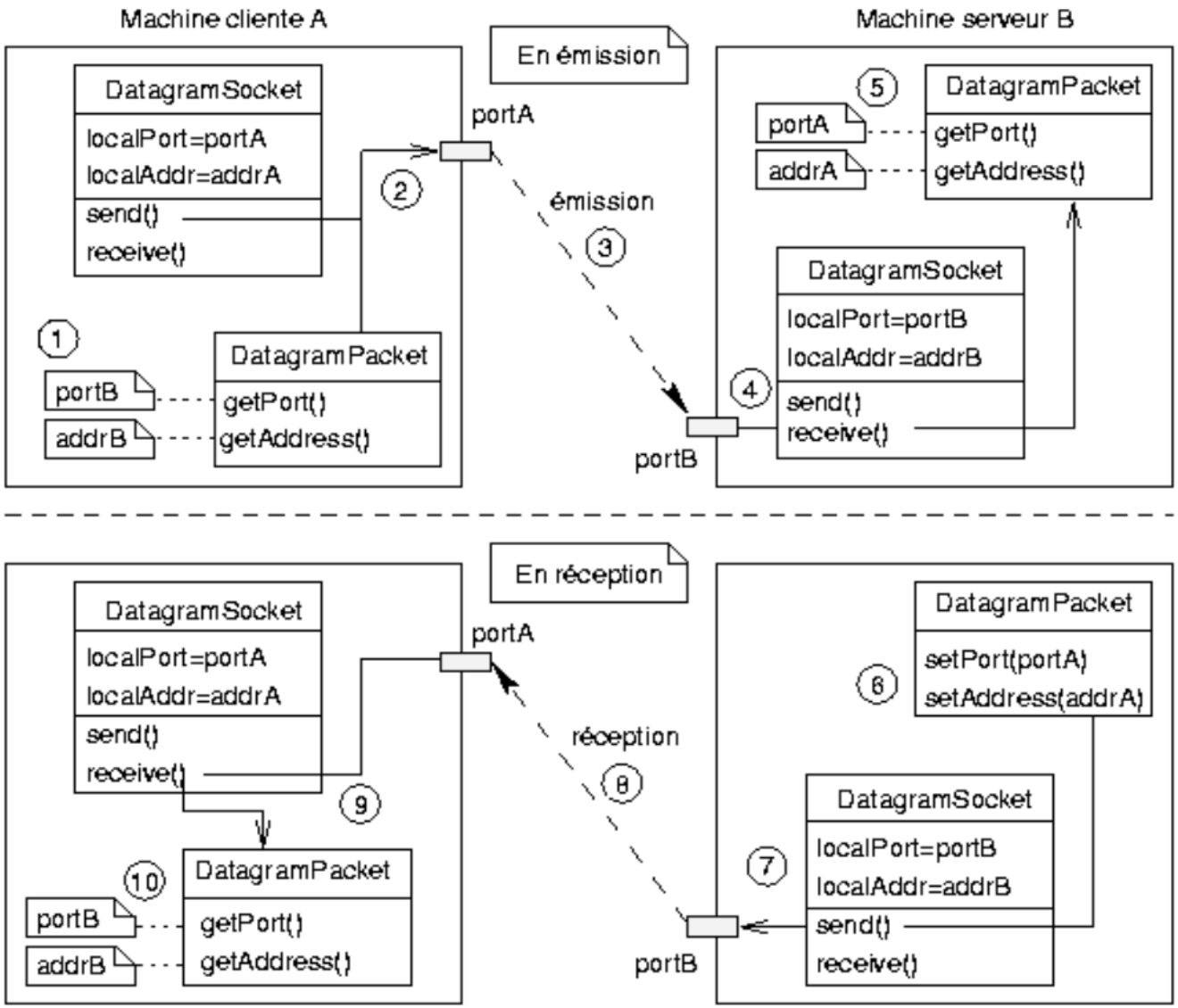
UDP avec Java - émission et réception

- Un `DatagramPacket` est fourni et pris en charge par un `DatagramSocket`.
- Pour émettre : `dSocket.send(dPacket) ;`.
- Pour recevoir : `dSocket.receive(dPacket) ;`.
  - Bloquant tant que rien n'est reçu !

UDP avec Java - DatagramPacket

- Différentes méthodes d'accès aux champs.
- `[set/get]Address()`, `[set/get]Port()`, `getSocketAddress()`
  - Concerne la machine distante (qui a émis ou va recevoir).
- `[set/get]Data()`, `getOffset()`, `setLength()`
  - Spécifie les données ou les bornes dans le tableau d'octets.
- `getLength()`
  - Normal en émission.
  - Avant réception : taille de la zone de stockage.
  - Après réception : nombre d'octets reçus.

UDP avec Java - Vue d'ensemble



UDP avec Java

- Données reçues au delà de la zone de stockage : perdues !
- Possibilité de limiter le temps d'attente en réception : `socket.setSoTimeout(int ms)` : lève une exception.

UDP avec Java - Exemple émission client

```
1  // socket sur port libre & adresse wildcard locale
2  DatagramSocket socket = new DatagramSocket() ;
3  //tab d'octets correspondant a la String ``Hello''
4  byte[] buf = "Hello".getBytes("ASCII") ;
5  //creation d'un datagramme contenant ces donnees
6  //destine au port 3333 de la machine de nom serveur
7  DatagramPacket packet =
8     new DatagramPacket(buf, buf.length, InetAddress.getByName("serveur"), 3333) ;
9  // envoi du datagramme via la socket
10 socket.send(packet) ;
```

UDP avec Java - Exemple réception client

```
1  //allocation & mise en place d'1 buffer pour recep.
2  byte[] receiveBuffer = new byte[1024] ;
3  packet.setData(receiveBuffer) ;
4  System.out.println(packet.getLength()) ;
5  // affiche : 1024 (taille de la zone de stockage)
6  // mise en attente de reception
7  socket.receive(packet) ;
8  System.out.println(packet.getLength()) ;
9  //affiche le nombre d'octets bien recus (<= 1024)
10 //construction d'une String correspondant aux octets recus
11 String s = new String(receiveBuffer, 0, packet.getLength(), "ASCII") ;
12 System.out.println(s) ;
13 //Quelle est la taille de la zone de stockage ici ?
```

UDP avec Java - réception serveur

- Pour un client : choix du numéro de port d'attache de la socket sans importance. Q : Pourquoi ?

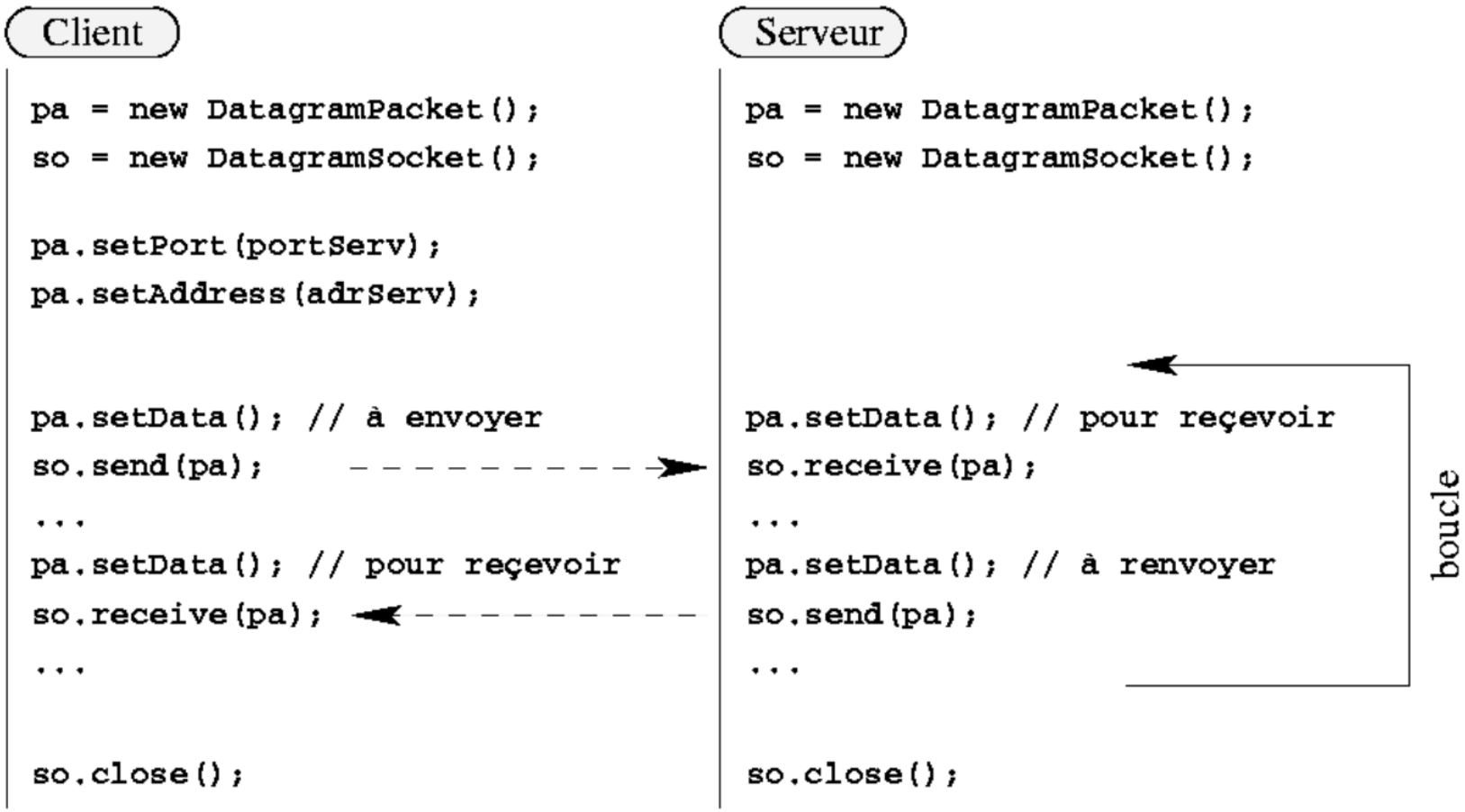
UDP avec Java - réception serveur

- Pour un client : choix du numéro de port d'attache de la socket sans importance. Q : Pourquoi ?
  - Connu par le serveur à la réception.
- Pour un serveur, doit être communiqué aux clients voulant communiquer !
  - Construire un DatagramSocket avec un port spécifique (risque de port non libre : exception !).

UDP avec Java - Exemple réception serveur

```
1  //socket d'attente de client, attachee au port 3333
2  DatagramSocket socket = new DatagramSocket(3333) ;
3  //datagramme pour la reception avec alloc. de buffer
4  byte[] buf = new byte[1024] ;
5  DatagramPacket packet=new DatagramPacket(buf,buf.length) ;
6  byte[] msg = "You're welcome !".getBytes("ASCII") ;
7  // message d'accueil
8  while (true) {
9   socket.receive(packet) ;//attente de reception bloquante
10  // place les donnees a envoyer
11  // (@ip et port distant sont deja ok)
12  packet.setData(msg) ;
13  socket.send(packet) ; // envoie la reponse
14  // replace la zone de reception
15  packet.setData(buf,0,buf.length) ;
```

UDP avec Java - client/serveur



Caractères et octets

- Ce qui est envoyé dans les paquets sont des *octets*.
- Si on veut envoyer du texte de A vers B, il faut coder/décoder.
  1. Coder le texte (chaine de caractères) en octets chez A.
  2. Envoi des octets de A vers B.
  3. Réception des octets par B.
  4. Décodage des octets en chaîne par B.

Caractères et octets - Java

- Conventions de conversions caractères/octets dans la classe `Charset`.
  - Par exemple : ASCII, UTF-8, Latin1...
  - Différences dans la manière de coder des accents, des lettres inusuelles...
  - Un caractère peut être codé par un ou plusieurs octets...

Caractères et octets - Java

- Conventions de conversions caractères/octets dans la classe `Charset`.
  - Par exemple : ASCII, UTF-8, Latin1...
  - Différences dans la manière de coder des accents, des lettres inusuelles...
  - Un caractère peut être codé par un ou plusieurs octets...
- Chaîne vers octets : `byte[] String.toBytes(String charset)`.
- Octets vers chaîne : `new String(byte[] b, int start, int end, String charset)`.

Caractères et octets - Java

- Conventions de conversions caractères/octets dans la classe `Charset`.
  - Par exemple : ASCII, UTF-8, Latin1...
  - Différences dans la manière de coder des accents, des lettres inusuelles...
  - Un caractère peut être codé par un ou plusieurs octets...
- Chaîne vers octets : `byte[] String.toBytes(String charset)`.
- Octets vers chaîne : `new String(byte[] b, int start, int end, String charset)`.
- Si pas de charset, charset par défaut (`static Charset Charset.defaultCharset()`).
- Charsets disponibles : `static SortedMap<String,Charset> Charset.availableCharsets()`.

Convertir en tableau d'octets

- La méthode peut exister (String...).
- On peut le faire "à la main".
- On peut utiliser les "nouvelles entrées sorties" java (NIO).
- Peut utiliser un stream sur le tableau d'octets.

```
1 ByteArrayOutputStream bais = new ByteArrayOutputStream() ;
2 DataOutputStream dos = new DataOutputStream(bais) ;
3 dos.writeInt(5) ;
4 dos.close() ;
5 byte[] buf = bais.toByteArray() ;
```

UDP avec Java - Pseudo-connexion (faux-ami)

- Si un serveur doit échanger plusieurs datagrammes consécutifs avec un même client, possibilité de ne considérer *que* ce client avec une *pseudo-connexion*.
- `connect(InetAddress,int)` ou `connect(SocketAddress)` pour établir la pseudo-connexion.
- `disconnect()` pour la terminer.
- Pendant ce temps, tous les datagrammes venant d'autres adresses/ports sont ignorés !

UDP - Broadcast, Multicast

- UDP permet la communication en diffusion.
- Efficace si le protocole de liaison sous-jacent offre la diffusion (ex. Ethernet).
  - Une seule trame est utilisée pour transporter un segment à plusieurs machines.

UDP - Broadcast, Multicast

- UDP permet la communication en diffusion.
- Efficace si le protocole de liaison sous-jacent offre la diffusion (ex. Ethernet).
  - Une seule trame est utilisée pour transporter un segment à plusieurs machines.
- Broadcast
  - De un à tous : un envoi à l'adresse 255.255.255.255 est destinée à toutes les machines du réseau local.
- Multicast
  - De un à plusieurs : un envoi à une adresse de classe D (224.0.0.0/4).
  - Destiné à toutes les machines qui se sont explicitement abonnées.

Cours 3 : Couche transport

Principes de la couche transport

Multiplexage et démultiplexage

Transport sans connexion : UDP

Transport avec connexion : TCP

TCP : Transmission Control Protocol (RFC 793)

(A) Salut, j'aimerai entendre une blague TCP.

(A) Salut, j'aimerai entendre une blague TCP.

(B) Salut, vous aimeriez entendre une blague TCP ?

(A) Oui, j'aimerai entendre une blague TCP.

(B) OK, je vais vous faire une blague TCP.

(A) OK, je vais écouter votre blague TCP.

(B) Etes-vous prêt à écouter une blague TCP ?

(A) Oui, je suis prêt à écouter une blague TCP.

(B) OK, je vais vous envoyer une blague TCP. Cela va durer 10 secondes, il y a 2 personnages, il n'y a pas de réglages et cela termine par une chute.

(A) OK, je suis prêt à recevoir votre blague TCP qui dure 10 secondes, a 2 personnages, n'a pas de réglages et se terminant par une chute.

(B) Désolé, votre connexion a été interrompue.

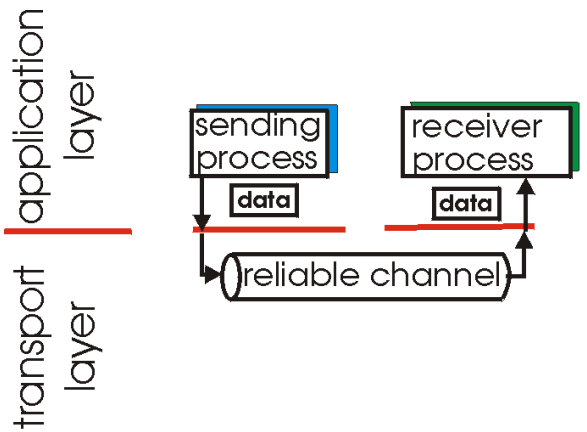
(B) Voulez-vous entendre une blague TCP ?

TCP : Transmission Control Protocol (RFC 793)

- Communication :
  - Par flux.
  - Fiable.
  - En mode connecté.
  - Full duplex (dans les deux sens).
- Données bufferisées, encapsulées dans des datagrammes IP.
  - Flux découpé en segments.
- Mécanisme de contrôle de flux
- Beaucoup plus lourd d'implantation que UDP.

Transport fiable

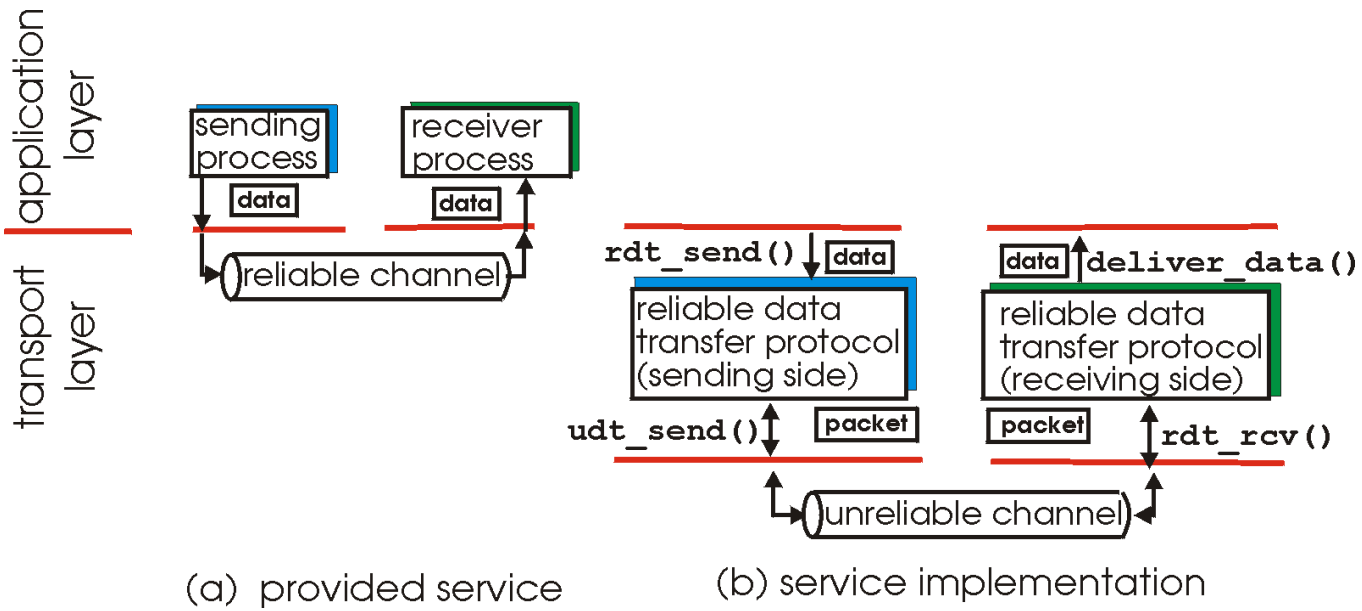
- Doit fournir un transport fiable entre deux processus.



(a) provided service

Transport fiable

- Doit fournir un transport fiable entre deux processus.
- Mais se base sur une couche sous-jacente non fiable !



Fiabilité

- Q : Comment les humains réparent les “erreurs” durant une conversation ?

Fiabilité

- Q : Comment les humains réparent les “erreurs” durant une conversation ?
- Acknowledgements (ACK) : receveur indique explicitement à l'envoyeur que ce qui est reçu est OK.
- Negative acknowledgements (NAK) : receveur indique explicitement à l'envoyeur qu'il y a une erreur dans ce qui est reçu.
  - Envoyeur renvoie.

Fiabilité

- Q : Que se passe t-il si les paquets ACK/NAK sont corrompus ?

Fiabilité

- Q : Que se passe t-il si les paquets ACK/NAK sont corrompus ?
  - L'envoyeur ne sait pas ce qu'il s'est passé !
  - Ne peut pas retransmettre : possibilité de doublons !

Fiabilité

- Q : Que se passe t-il si les paquets ACK/NAK sont corrompus ?
  - L'envoyeur ne sait pas ce qu'il s'est passé !
  - Ne peut pas retransmettre : possibilité de doublons !
- Ajout d'un numéro de séquence pour éviter les doublons (receveur supprime s'il l'a déjà).
- Mécanisme d'attente.

TCP - Segments

- Fiabilité obtenue par un mécanisme d'**acquittement des segments**.
  - À l'émission d'un segment, une alarme est amorcée.

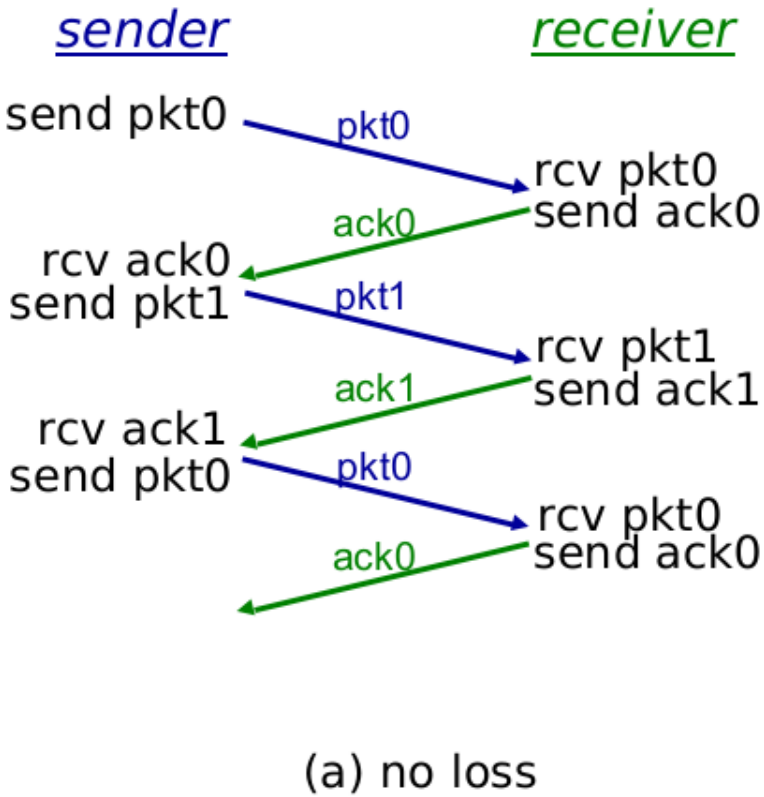
TCP - Segments

- Fiabilité obtenue par un mécanisme d'**acquittement des segments**.
  - À l'émission d'un segment, une alarme est amorcée.
  - Désamorcée que si l'acquittement correspondant est reçu.
  - Si expiration, segment réémis.

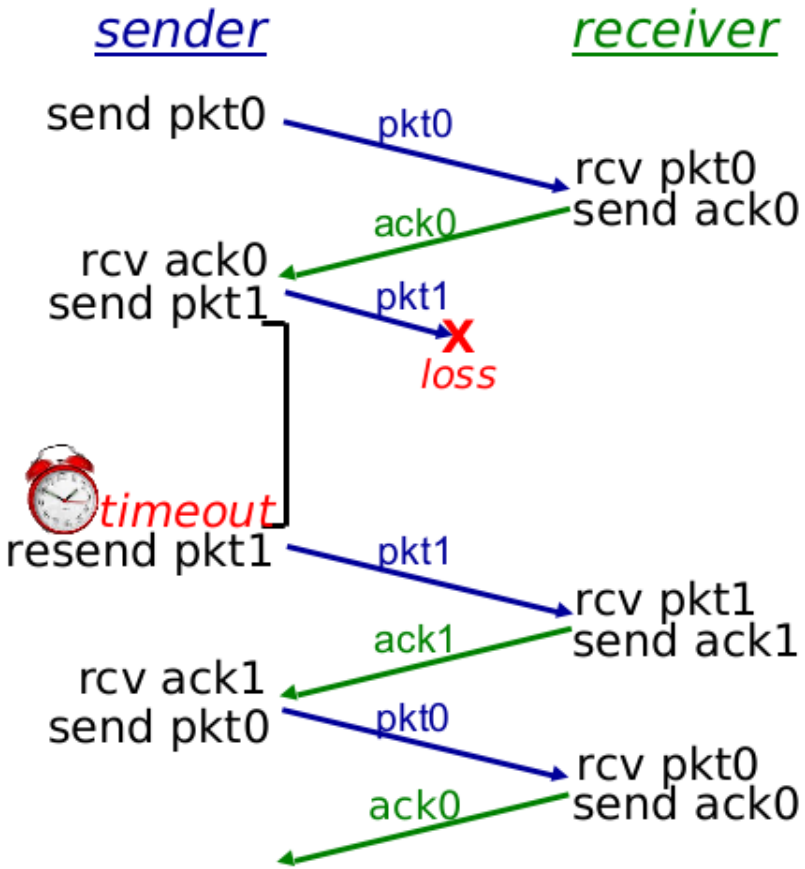
TCP - Segments

- Fiabilité obtenue par un mécanisme d'**acquittement des segments**.
  - À l'émission d'un segment, une alarme est amorcée.
  - Désamorcée que si l'acquittement correspondant est reçu.
  - Si expiration, segment réémis.
- Chaque segment possède un **numéro de séquence**.
  - Préserve l'ordre.
  - Évite les doublons.
  - Acquittements identifiés par un marqueur ACK.
  - Transport dans un même segments des donnes et de l'acquittement des données précédentes : piggybacking.

TCP - Echanges stop and wait

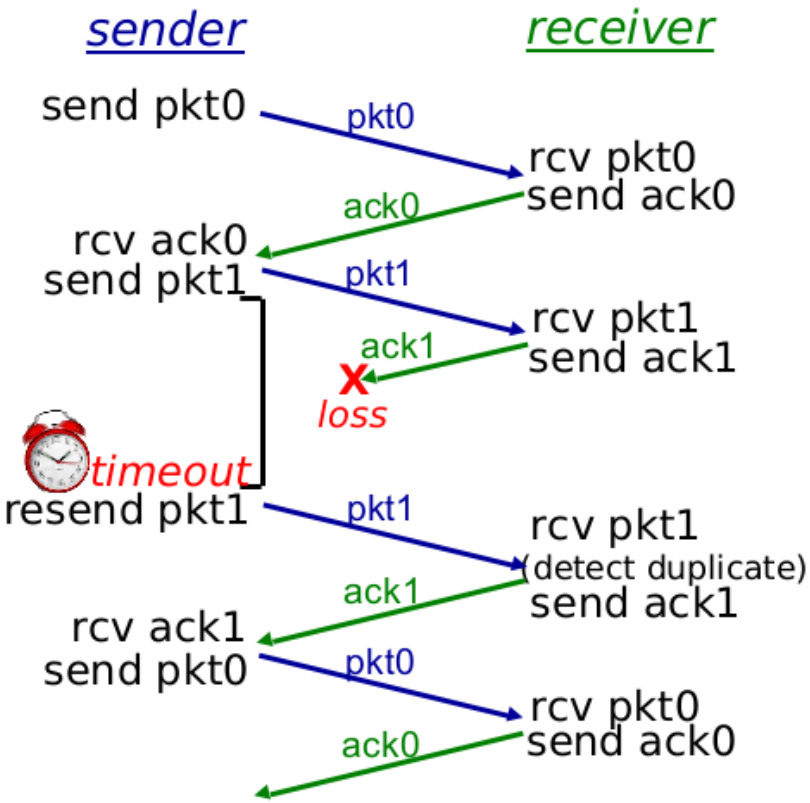


(a) no loss

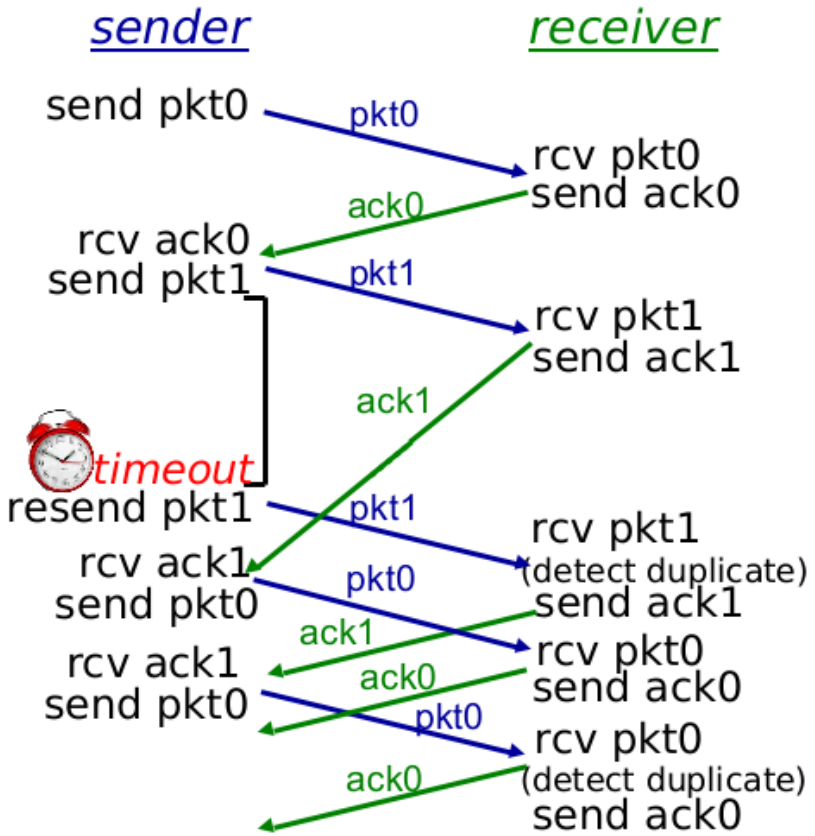


(b) packet loss

TCP - Echanges stop and wait



(c) ACK loss



(d) premature timeout/ delayed ACK

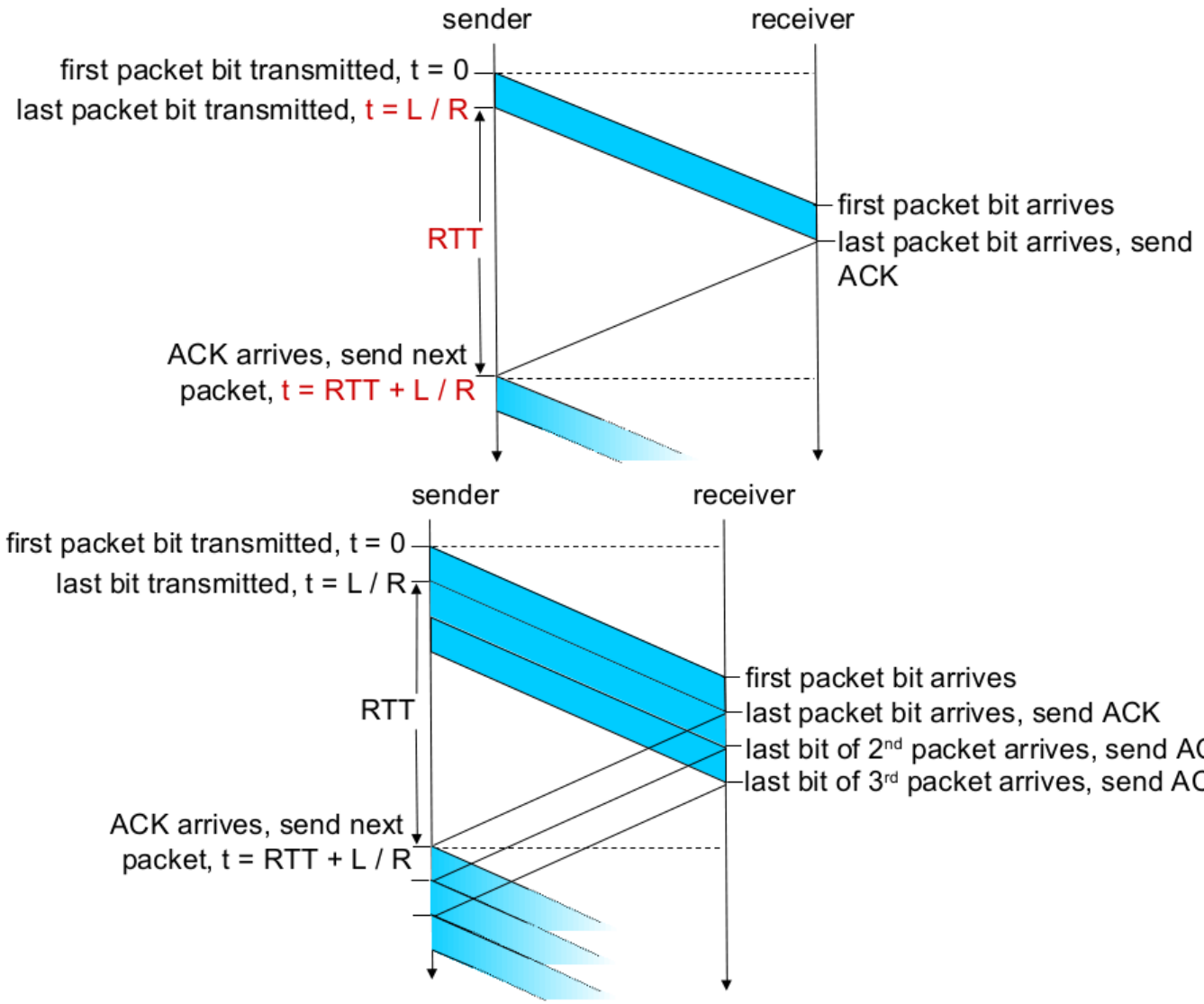
TCP - Echanges

- En pratique, beaucoup trop long !
- Doit attendre le retour (peut traverser la terre...) pour chaque paquet !
- Q : Peut mieux faire ?

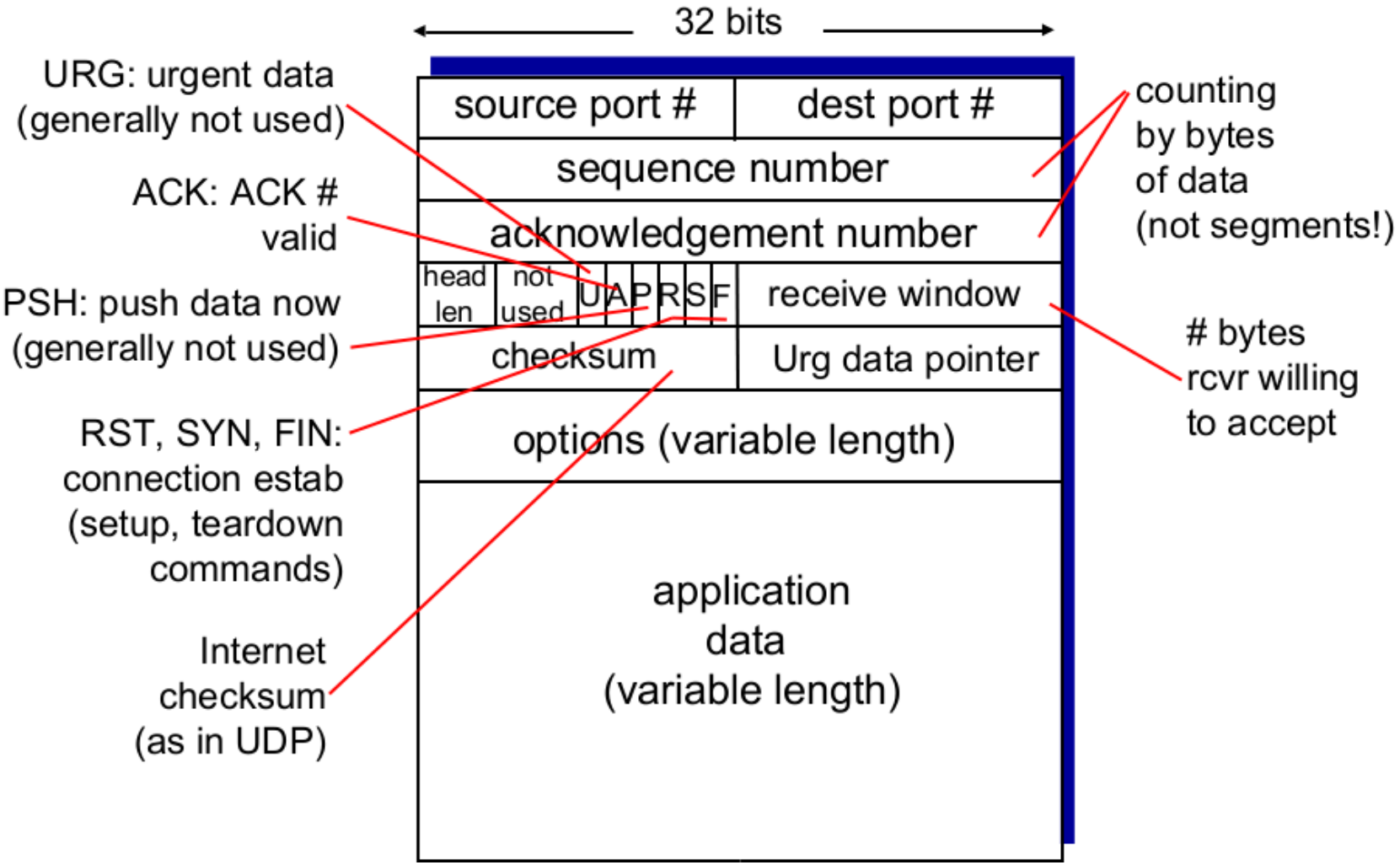
TCP - Echanges

- En pratique, beaucoup trop long !
- Doit attendre le retour (peut traverser la terre...) pour chaque paquet !
- Q : Peut mieux faire ?
- “Pipelinage”.
- Envoi de plusieurs paquets d’un coup, reçoit les ACK au fur et à mesure.
- Re-envoi des paquets non ACK.

TCP - Echanges wait vs pipelinage



TCP - Format

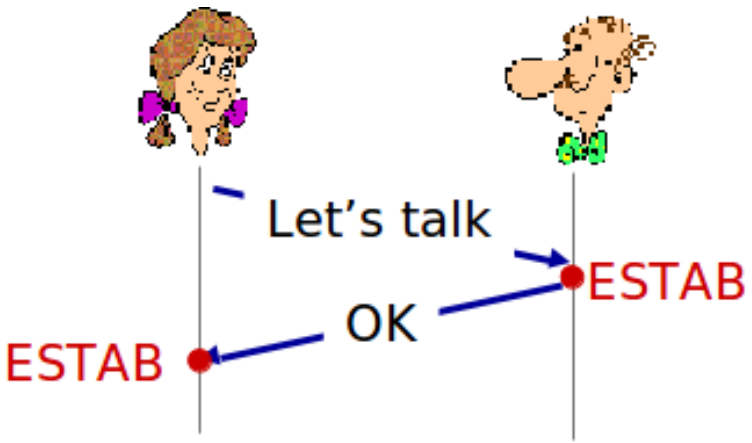

32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)
URG: urgent data (generally not used)
ACK: ACK # valid
PSH: push data now (generally not used)
RST, SYN, FIN: connection estab (setup, teardown commands)
Internet checksum (as in UDP)
counting by bytes of data (not segments!)
bytes rcvr willing to accept

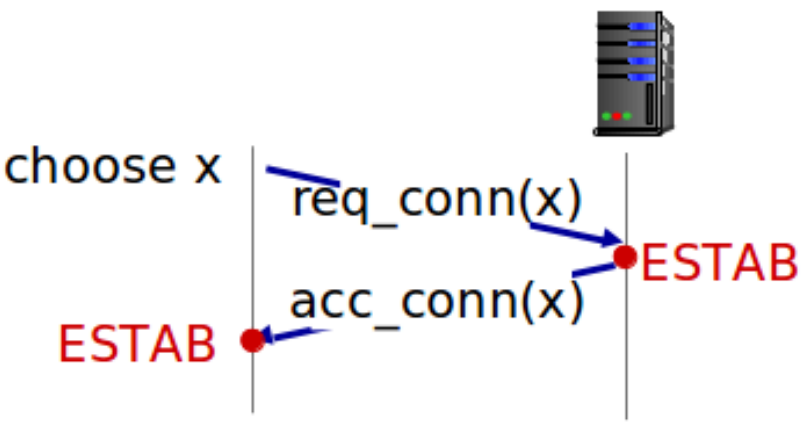
Etablissement de connexion

- Avant de commencer à échanger, “handshake” entre le client et le serveur.
- Agrément de la connexion (chacun sait que l'autre veut établir une connexion).

Etablissement de connexion - 2-way handshake

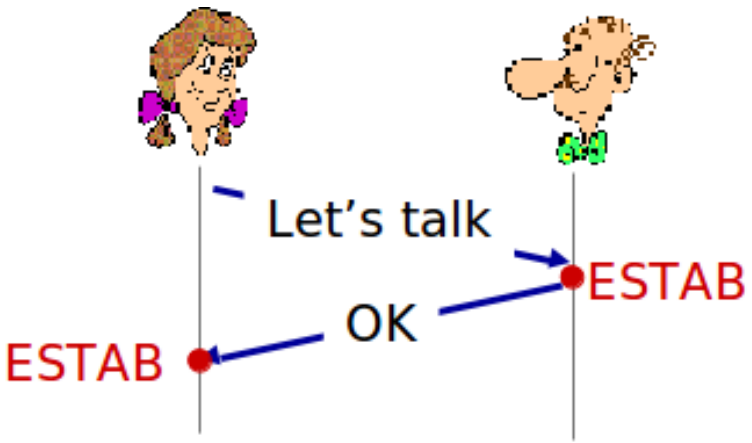
- x est un identifiant unique.

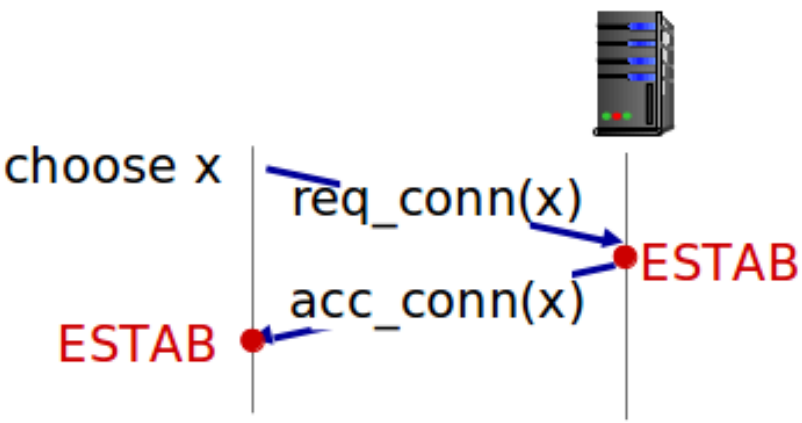


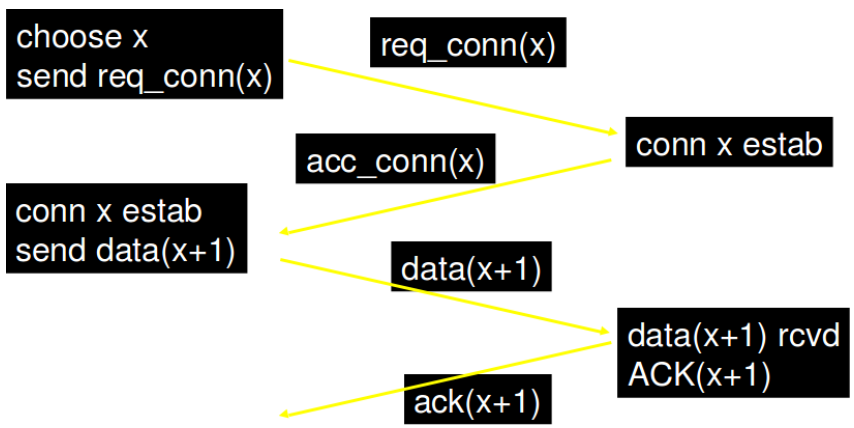


Etablissement de connexion - 2-way handshake

- x est un identifiant unique.





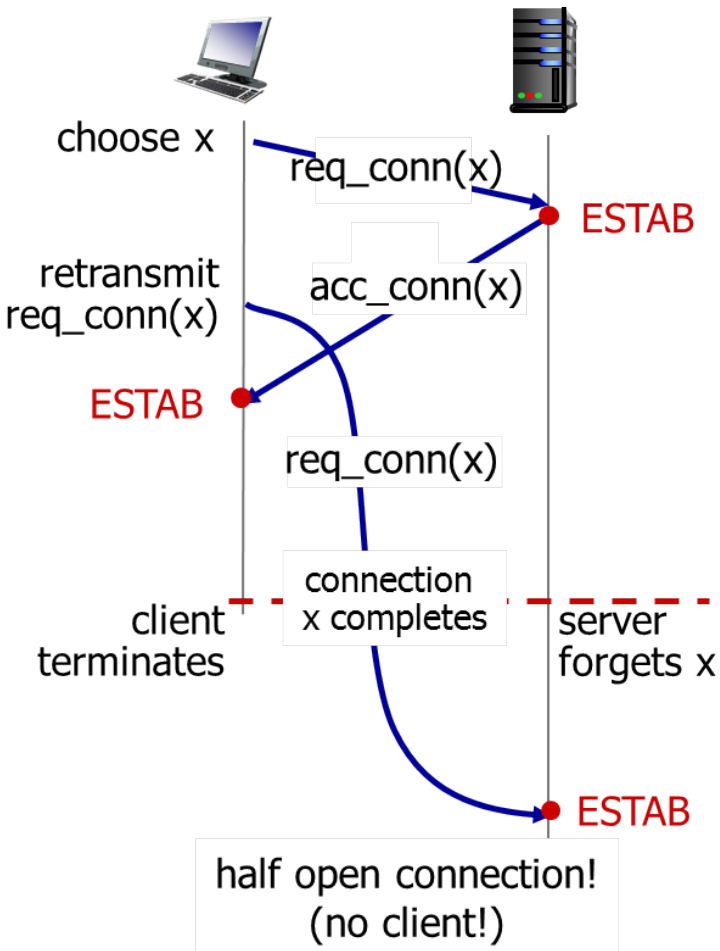


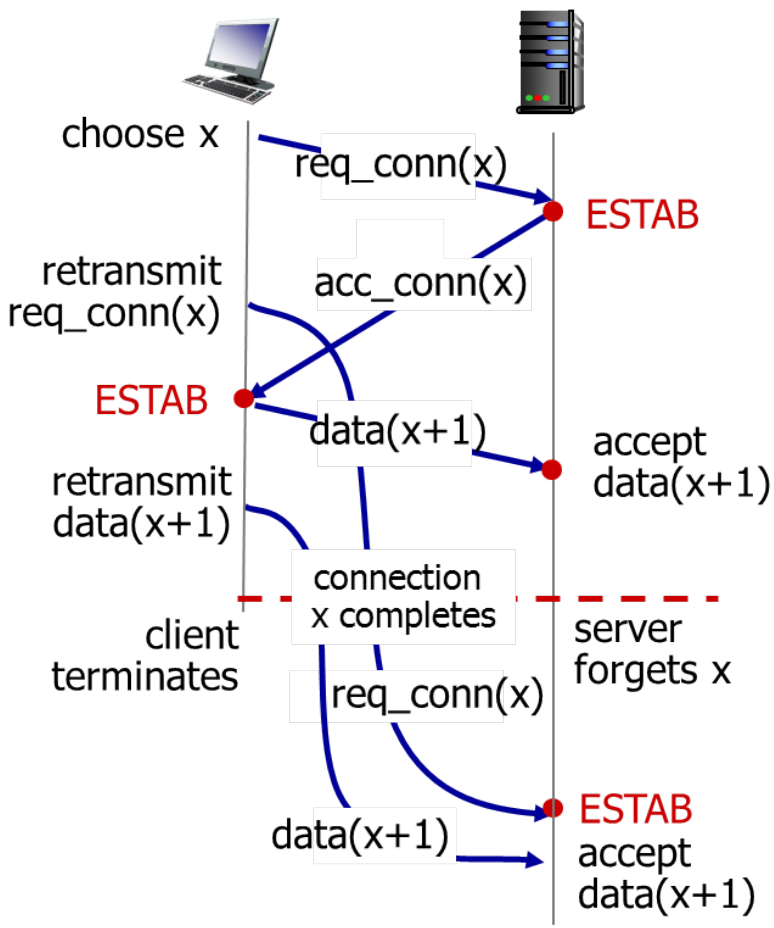
Etablissement de connexion - 2-way handshake

- Possibilité de dysfonctionnement.
  - Délais variables.
  - Messages retransmis (par ex. $req\_connex(x)$ car perdu).
  - ...

Etablissement de connexion - 2-way handshake

- Scénarios d'échec.





Serveur ne peut reconnaitre si $req\_conn(x)$ est un dupliqué.

Etablissement de connexion TCP - 3-way handshake

- Client TCP envoie un segment spécial (SYN) au serveur, sans donnée, mais avec un bit d'en-tête spécial positionné à 1 (SYN). Choisit et envoie un numéro de séquence initial x.

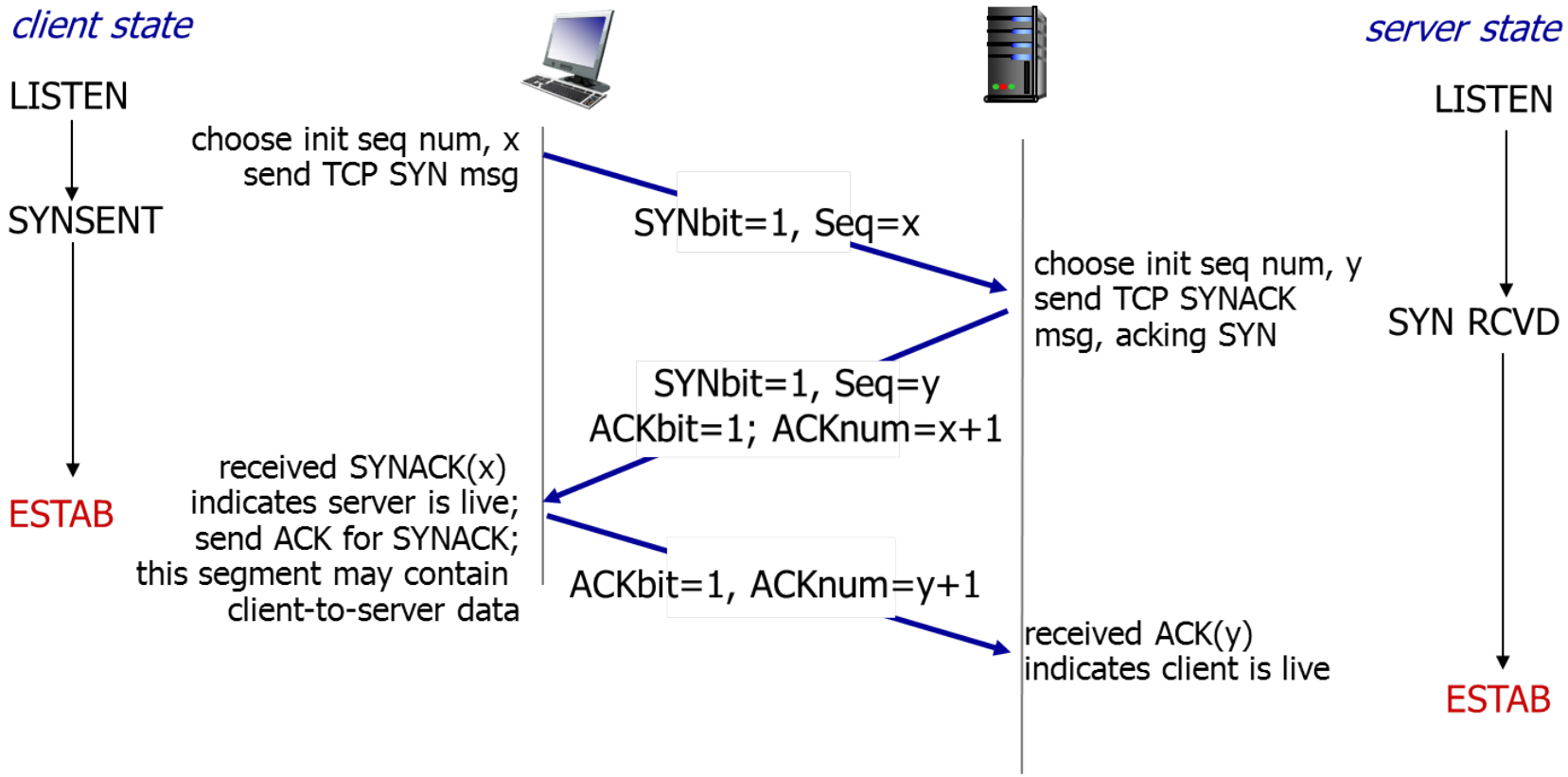
Etablissement de connexion TCP - 3-way handshake

- Client TCP envoie un segment spécial (SYN) au serveur, sans donnée, mais avec un bit d'en-tête spécial positionné à 1 (SYN). Choisit et envoie un numéro de séquence initial x.
- Serveur reçoit le SYN, alloue des tampons, des variables, et accuse réception du segment spécial avec un segment spécial, sans données. Bit SYN à 1 aussi, ACK à 1, ACK number à $x + 1$ et son propre numéro de séquence initial y.
  - Ce segment (SYNACK) pourrait se traduire par "J'ai bien reçu votre demande avec numéro de séquence initial x, je donne mon accord, et voici mon propre numéro de séquence initial.

Etablissement de connexion TCP - 3-way handshake

- Client TCP envoie un segment spécial (SYN) au serveur, sans donnée, mais avec un bit d'en-tête spécial positionné à 1 (SYN). Choisit et envoie un numéro de séquence initial x.
- Serveur reçoit le SYN, alloue des tampons, des variables, et accuse réception du segment spécial avec un segment spécial, sans données. Bit SYN à 1 aussi, ACK à 1, ACK number à $x + 1$ et son propre numéro de séquence initial y.
  - Ce segment (SYNACK) pourrait se traduire par "J'ai bien reçu votre demande avec numéro de séquence initial x, je donne mon accord, et voici mon propre numéro de séquence initial.
- Client reçoit le SYNACK, alloue aussi tampons et variables. Envoie un autre segment au serveur accusant réception, SYN à 0 (connexion établie), ACK à 1 et ACK number à $y + 1$.

Etablissement de connexion TCP - 3-way handshake



3-way handshake

- Possibilité (mid '90s) d'attaque par déni de service (SYN flood).
- Déluge d'étape 1 (SYN), mais jamais l'étape 3 côté client.
- Le serveur prépare la connexion (allocation de ressources), mais ne recoit jamais la dernière étape de validation.
- Il devient surchargé, un client normal n'obtient pas l'étape 2.

3-way handshake

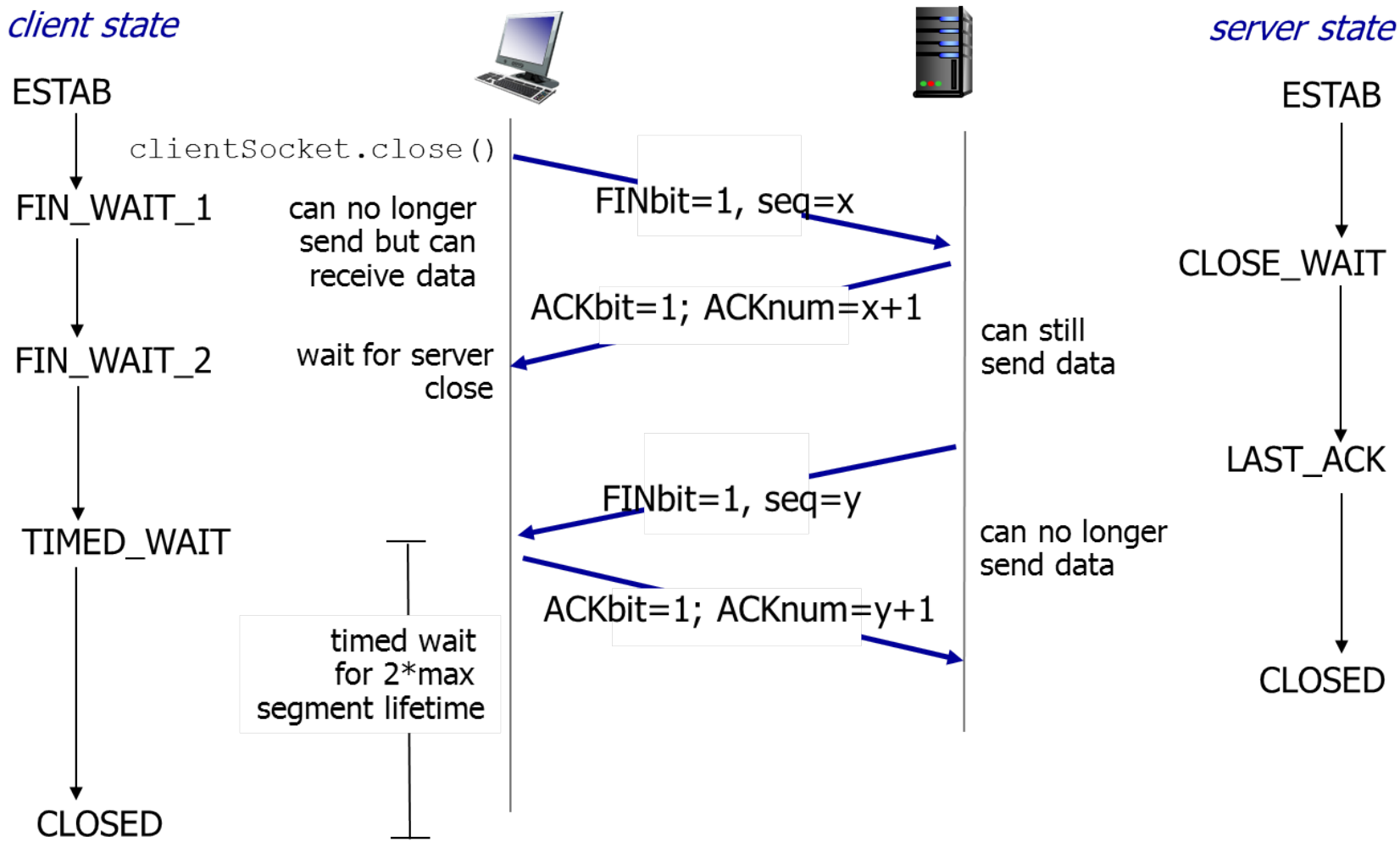
- Possibilité (mid '90s) d'attaque par déni de service (SYN flood).
- Déluge d'étape 1 (SYN), mais jamais l'étape 3 côté client.
- Le serveur prépare la connexion (allocation de ressources), mais ne recoit jamais la dernière étape de validation.
- Il devient surchargé, un client normal n'obtient pas l'étape 2.

- Parades possibles (éviter d'allouer des ressources tant que connexion pas entièrement établie...).

Fermeture de connexion TCP

- Client (ou serveur) décide de terminer la connexion.
- Client envoie un segment avec le bit FIN à 1.
- Attend l'ACK sur serveur.
- Attend la réception d'un bit FIN du serveur.
- Envoi d'un segment pour en spécifier sa réception.

Fermeture de connexion TCP



Java et TCP - Sockets

- `java.net.ServerSocket`
  - Côté serveur, représente l'objet socket en attente d'une connexion client.

Java et TCP - Sockets

- `java.net.ServerSocket`
  - Côté serveur, représente l'objet socket en attente d'une connexion client.
- `java.net.Socket`
  - Représente une connexion TCP.
    - Côté client, on crée la connexion via sa création.
    - Côté serveur, l'acceptation via une ServerSocket retourne cet objet.

Java et TCP - Sockets

- `java.net.ServerSocket`
  - Côté serveur, représente l'objet socket en attente d'une connexion client.
- `java.net.Socket`
  - Représente une connexion TCP.
    - Côté client, on crée la connexion via sa création.
    - Côté serveur, l'acceptation via une ServerSocket retourne cet objet.
- Comme pour UDP, attachée à une adresse IP et un port.

Java et TCP - Sockets client

- En TCP, les deux sockets vont communiquer.
- Construire un objet `Socket`, éventuellement l'attacher localement et la connecter à la socket d'un serveur.

Java et TCP - Sockets client

- En TCP, les deux sockets vont communiquer.
- Construire un objet `Socket`, éventuellement l'attacher localement et la connecter à la socket d'un serveur.
- `Socket()`
  - Puis `bind(SocketAddress bindPoint)` pour attachement local
  - `connect(SocketAddress sa)` ou `connect(SocketAddress sa, int timeOut)` pour établir la connexion.
- `Socket(InetAddress addr, int port)`
- `Socket(String host, int port)`
- `Socket(InetAddress addr, int port, InetAddress localAddr, int localPort)`
- `Socket(String host, int port, InetAddress localAddr, int localPort)`

Java - Attachement

- Si `bind()` avec `null` : choisit une adresse valide et un port libre.
- Échec possible si port déjà occupé ou socket déjà attachée (testable) : lève des exceptions.

Java - Attachement

- Si `bind()` avec `null` : choisit une adresse valide et un port libre.
- Échec possible si port déjà occupé ou socket déjà attachée (testable) : lève des exceptions.
- Lors du `connect()`, socket attachée automatiquement si pas déjà fait (handshake).
- Bloquant, retourne normalement si succès.

Java - Utilisation de la socket

- Connexion établie une fois l'objet socket construit, attaché et connecté.
- `OutputStream getOutputStream()` donne un flux d'écriture sur la connexion.
- `InputStream getInputStream()` donne un flux de lecture des données arrivant sur la connexion.
- Méthodes de lecture bloquantes.
  - Timeout possible (`setSoTimeout(int ms)`), exception au delà.

Java - Fermeture de socket

- `close()`, `isClosed()`
  - Fermer un flux ferme les 2 sens de la communication.
  - Non bloquant, mais reste ouvert tant qu'il y a des données.
  - `setSoLinger` rend `close()` bloquant.
- Possibilité de fermer qu'un seul sens.
  - `socket.shutdownOutput()`, `isOutputShutdown()`
    - Handshake de déco une fois toutes les données présentes émises.
    - Exception si tentative d'écriture.
  - `socket.shutdownInput()`, `isInputShutdown()`
    - javadoc

Java - Exemple client

```
1  //Creation de l'objet socket et connexion
2  Socket s = new Socket(serverName, port) ;
3  //Affichage des extremites de la connexion
4  System.out.println("Connexion entre " + s.getLocalSocketAddress() + " et " + s.
       getRemoteSocketAddress()) ;
5  // Recup. d'1 flot en ecriture et envoi de donnees
6  PrintStream ps = new PrintStream(s.getOutputStream(),"ASCII") ;
7  ps.println("Hello !") ;
8  //Recup. d'1 flot en lecture et recep. de donnees
9  //(1 ligne)
10 BufferedReader br = new BufferedReader(new InputStreamReader(s.getInputStream(),"
       ASCII")) ;
11 String line = br.readLine() ;
12 System.out.println("Donnees recues  : " + line) ;
13 s.close() ;
```

Java - Socket serveur

- `java.net.ServerSocket`
  - *Attendre* les connexions des clients.
- `ServerSocket()` puis `bind`.
- `ServerSocket(int port)`, `ServerSocket(int port, int queue)`, `ServerSocket(int port, int queue, InetAddress addr)`.
  - Si port 0, attaché à un port libre (doit être divulgué aux clients).
  - queue : taille de la queue, connexions "en attente" ...

Java - Accepter

- `Socket s = serverSocket.accept()`.
  - Bloquant ! Timeout possible (`setSoTimeout(int ms)`)
- La socket retournée est dite *de service* et représente la connexion établie.
- On peut récupérer adresses et ports (de chaque côté) depuis cet objet.

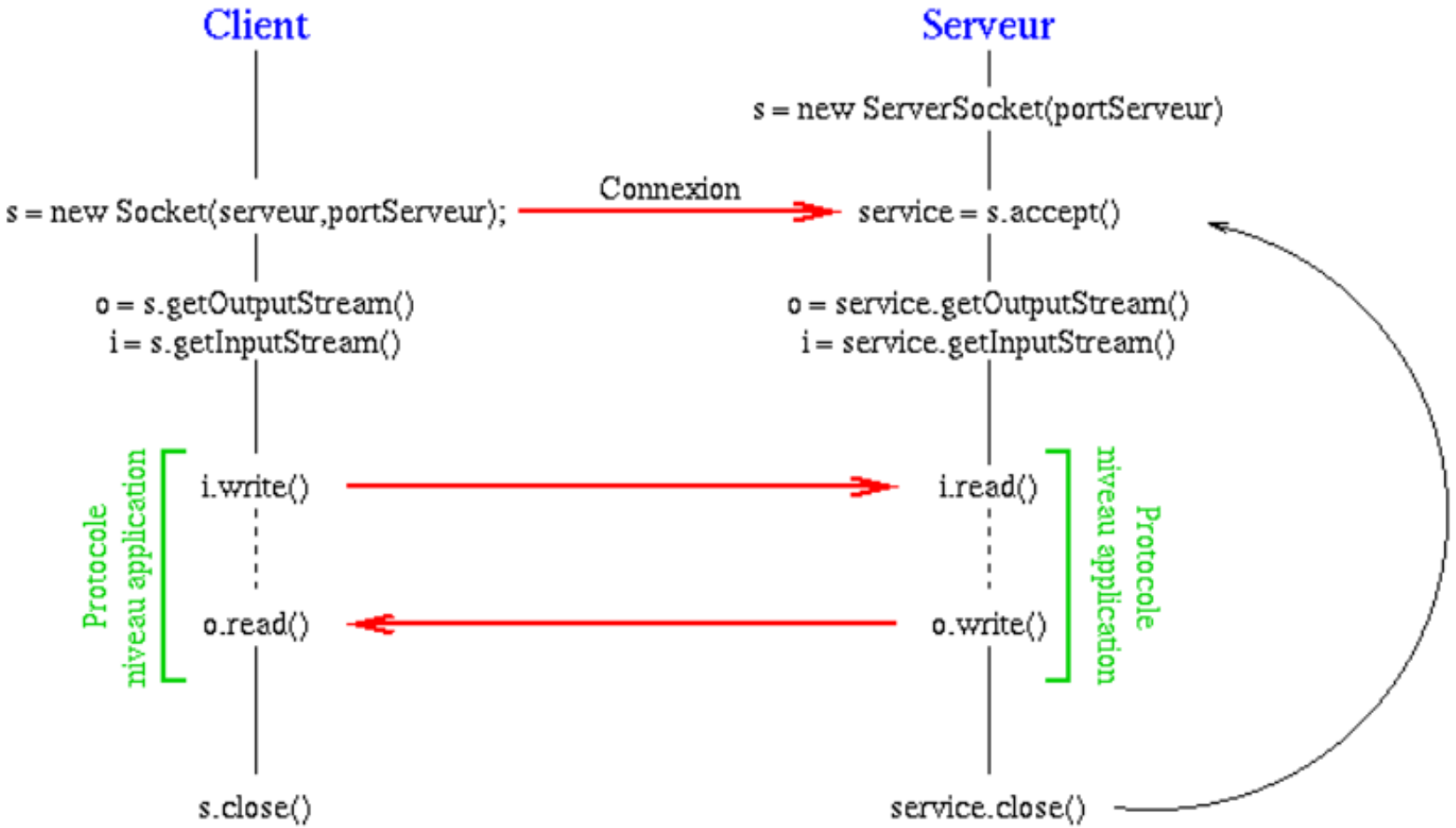
Java - Accepter

- `Socket s = serverSocket.accept()`.
  - Bloquant ! Timeout possible (`setSoTimeout(int ms)`)
- La socket retournée est dite *de service* et représente la connexion établie.
- On peut récupérer adresses et ports (de chaque côté) depuis cet objet.
- Si plusieurs sockets sont retournées par accept du même objet ServerSocket, ils ont le même port et la même IP locale.
  - Démultiplexage avec adresse et port client.

Java - Exemple de serveur inutile

```
1  //Creation et attachement d'un objet ServerSocket
2  ServerSocket ss = new ServerSocket(3333) ;
3  while (true) {
4    Socket s = ss.accept() ;//mise en attente de connex.
5    // Recup. des flots lecture/ecriture
6    BufferedReader br = new BufferedReader(new InputStreamReader(s.getInputStream(),
           "ASCII")) ;
7    PrintStream ps = new PrintStream(s.getOutputStream(), true, "ASCII") ;
8    System.out.println("Recu : "+br.readLine()) ;
9    //Affiche le "Hello" recu et envoie //inlassablement le meme message
10   ps.println("You're welcome !") ;
11   s.close() ;
12   // Ferme la socket "de service", pour en accepter une autre
13 }
```

Java - Schéma connexion TCP



Java - Lire

- Depuis une socket, on peut récupérer un `InputStream`.
  - Accès au flux *entrant* sur la socket pour y *lire* des données.
- `int read(byte[] b, [int offset, int len])` : remplissage du tableau b avec le contenu du flux. Retourne le nombre d'octets lus, -1 en fin de flux. Bloquant.
- `void close()` : libération du flux.

- Opération bloquante : exceptions doivent être gérées.

Java - Ecrire

- Depuis une socket, on peut récupérer un `OutputStream`.
  - Accès au flux *sortant* sur la socket pour y *écrire* des données.
- `void write(byte[] b, [int offset, int len])` : écriture du tableau b sur le flux.
- `void close()` : vide le tampon (`flush()`) puis libération du flux.

Java - Lire et écrire : classes utiles

- Classes dérivant de {Input,Output}Stream pour une gestion simplifiée des flux.
- File{Input,Output}Stream : lecture/écriture d'un fichier.
- Data{Input,Output}Stream : encapsule un {Input,Output}Stream et permet d'y lire/écrire des types primitifs comme int, long,...
- ...

Java - Lire et écrire des flux de caractères

- Flux de caractères : flux binaire + codage octets/caractères (charset).
- Classes dérivant de `Reader`,`Writer`, implantant les mêmes méthodes que les streams binaires mais avec des `char` à la place de `byte`.

Java - Lire et écrire des flux de caractères

- `{Input,Output}Stream{Reader,Writer}` : construction depuis un `{Input,Output}Stream` + un Charset.
- `Buffered{Reader,Writer}` : ajout d'une bufferisation. Permet par exemple de lire une ligne entière avec `readLine()`.

TCP - Gérer plusieurs clients

- Socket d'écoute du serveur.
- A la connexion d'un client, socket de service pour communiquer avec le client.
- Envoi de données :
  - En UDP, on précise l'adresse du client dans le paquet à envoyer.
  - En TCP, on utilise la socket de service.
- Réception de données :
  - En UDP, attente d'un paquet.
  - En TCP, attente de données sur toutes les sockets actives.

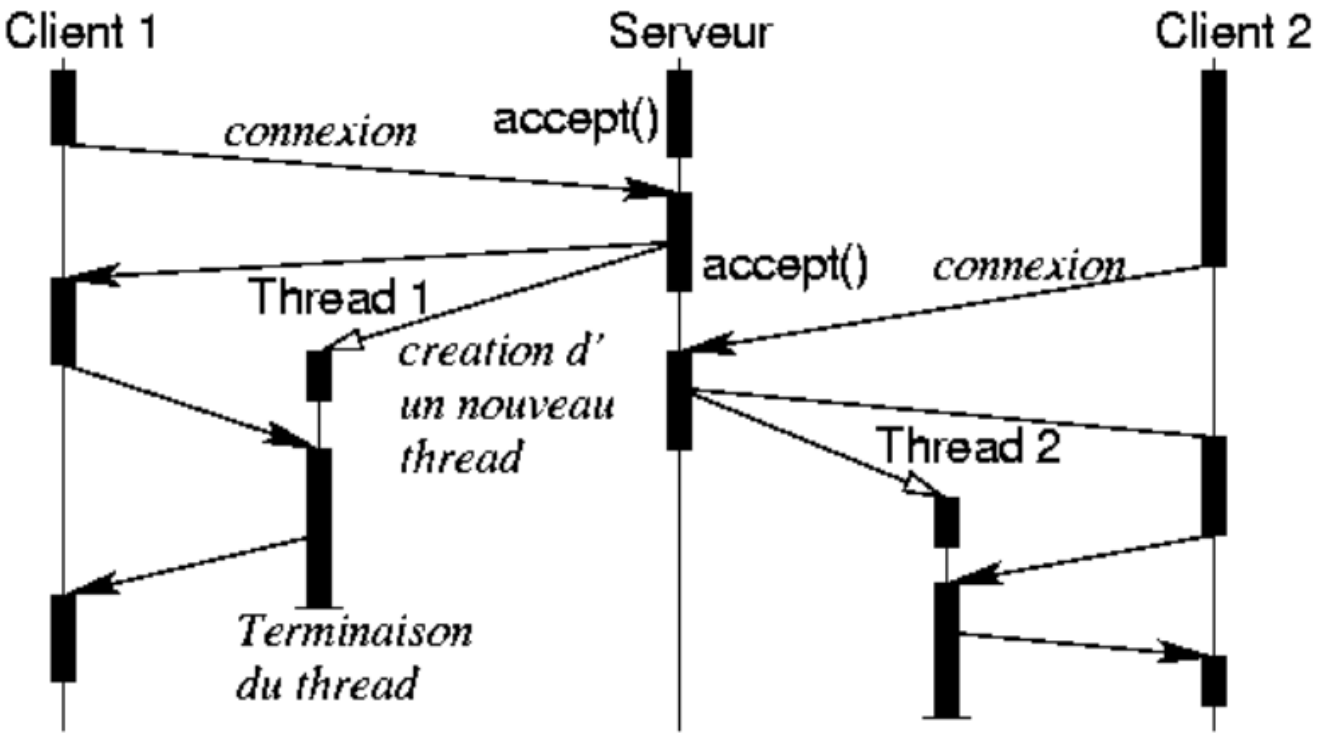
TCP - Gérer plusieurs clients

- Problèmes :
  - Attente de connexions sur la socket serveur : **bloquant** jusqu'à la prochaine connexion.
  - Attente de lecture sur une socket de service : **bloquant** jusqu'à réception de données.

TCP - Gérer plusieurs clients

- Problèmes :
  - Attente de connexions sur la socket serveur : **bloquant** jusqu'à la prochaine connexion.
  - Attente de lecture sur une socket de service : **bloquant** jusqu'à réception de données.
- Impossible à effectuer avec un seul processus !
- Avec des **threads** :
  - Un thread en attente sur la socket serveur.
  - Un thread pour chaque nouveau client (attention !).

TCP - Gérer plusieurs clients



Java et TCP

- Peut faire avec un seul thread en utilisant du non-bloquant (nio)...
- Compliqué !
- Idée :
  - Ouvrir des **canaux** (channels) en mode non-bloquant (lecture/écriture retourne immédiatement même si rien à lire/écrire).
    - Représente des opérations d'IO sur des fichiers/sockets/tubes...
  - Utiliser un **selecteur** pour attendre en même temps les entrées-sorties sur les canaux.
    - Attend passivement et retourne (appel système) quand il détecteur des IO sur les canaux.
    - Besoin d'un unique thread !

Contrôle de flux

- Buffer de réception sur chaque hôte stockant les données qui arrivent.
- L'application récupère les données qui lui sont destinées.
- Mais, l'application peut être occupée et prendre un certain temps avant de récupèrer les données.
- Sans contrôle, l'expéditeur pourrait saturer le buffer en envoyant à un rythme trop soutenu.

Contrôle de flux

- Le destinataire “informe” sur l’espace libre dans son buffer avec un champ dans l’en-tête des paquets TCP.
- L’expéditeur limite alors ses envois en fonction.

▸ Applet

Contrôle de flux

- Le destinataire “informe” sur l’espace libre dans son buffer avec un champ dans l’en-tête des paquets TCP.
- L’expéditeur limite alors ses envois en fonction.
- UDP ne le fait pas : les segments non lus sont perdus.

▸ Applet

Contrôle de congestion

- Différent du contrôle de flux (niveau réseau et non hôtes).
- Pertes : souvent dû à un trafic fort et à la saturation des buffers des routeurs.
- Remède : Retransmission de paquets.

Contrôle de congestion

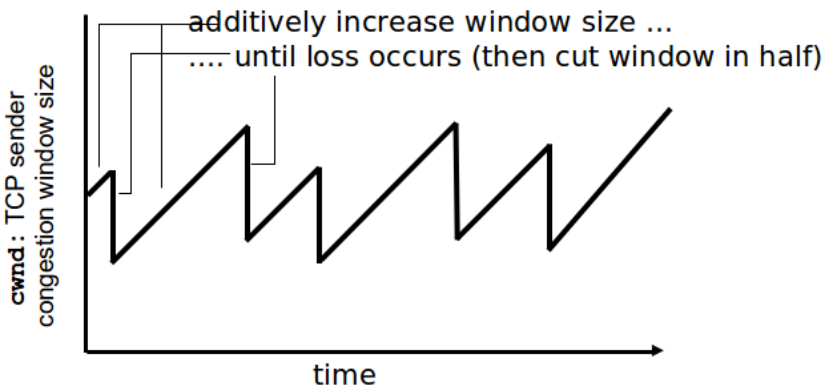
- Différent du contrôle de flux (niveau réseau et non hôtes).
- Pertes : souvent dû à un trafic fort et à la saturation des buffers des routeurs.
- Remède : Retransmission de paquets.
- Mais ne touche pas à la cause !
- Mécanisme de “bridage” des sources.

Contrôle de congestion - Approches

- Contrôle de congestion “bout en bout” (sans assistance de la couche réseau).
  - Congestion reconnue par les terminaux (surveillance sur perte ou retard).
  - Perte de segment $\rightarrow$ réduction de la taille des envois.
  - Adopté par TCP (car IP ne donne pas d'infos. sur la congestion du réseau).
- Contrôle de congestion assisté par le réseau.
  - Routeurs procurent des infos aux expéditeurs.
  - Réseaux SNA d'IBM, DECnet de DEC...

Contrôle de congestion - Approche TCP

- L'envoyeur augmente de manière additive la taille des segments.
- Si une perte arrive, division par deux de la taille courante !



Applet

Conclusion

- Principes de la couche transport.
  - Multiplexage/démultiplexage.
  - Transfert fiable.
  - ...
- UDP et TCP. Implémentation en Java.

Conclusion

- Principes de la couche transport.
  - Multiplexage/démultiplexage.
  - Transfert fiable.
  - ...
- UDP et TCP. Implémentation en Java.
- Couches 3, 2, 1 du modèle vues en Licence (cœur du réseau ($\neq$ périphérie avec opérations sur les terminaux)).

Conclusion

http://www.quizzoodle.com/session/
c6bd7265629445d7807f89e58d4ff72a