

# Couche de transport (UDP, transport fiable et TCP)

La couche applicative repose sur la couche de transport. Cette dernière s'en fout du type de donnée utilisée, cette couche a seulement pour but de transférer les données.

Il existe deux protocoles, le **TCP** (Transmission Control Protocol) qui permet d'envoyer des informations de manière fiables (en vérifiant la bonne réception des "paquets" de données), et l'**UDP** (User Datagram protocol) est un protocole qui envoie les paquets sans se soucier de la bonne réception. Ce dernier, bien que moins fiable, est beaucoup plus rapide.



**Kirk Bater**

@KirkBater

Follow



This image is a TCP/IP Joke. This tweet is a UDP joke. I don't care if you get it.

### Thread

iamkirkbater and jkjustjoshing



**iamkirkbater** 🌐 Aug 23rd, 2017 at 9:37 AM  
in #www

Do you want to hear a joke about TCP/IP?



7 replies



**jkjustjoshing** 5 months ago  
Yes, I'd like to hear a joke about TCP/IP



**iamkirkbater** 🌐 5 months ago  
Are you ready to hear the joke about TCP/IP?



**jkjustjoshing** 5 months ago  
I am ready to hear the joke about TCP/IP



**iamkirkbater** 🌐 5 months ago  
Here is a joke about TCP/IP.



**iamkirkbater** 🌐 5 months ago  
Did you receive the joke about TCP/IP?



**jkjustjoshing** 5 months ago  
I have received the joke about TCP/IP.



**iamkirkbater** 🌐 5 months ago  
Excellent. You have received the joke about TCP/IP. Goodbye.

Il faut donc connaître le port du programme à contacter, pour cela le système maintient un annuaire liant un numéro de port à une application.

## L'UDP (User Datagram Protocol)

40	320	Source Port	Destination Port
44	352	Length	Checksum
48	384+	Data	

Avec l'UDP on va simplement transmettre les données sans se soucier de leur bonne réception. Ainsi pour chaque message (TPDU, Transport Protocol Data Unit) il faut connaître le numéro de port source (et destination ainsi que la longueur du message et éventuellement un "checksum" permettant de vérifier l'intégrité des informations.

Le protocole UDP est très utilisé pour les applications qui ont besoin d'aller vite, même si cela veut dire de potentiellement perdre des informations. Par exemple pour les jeux massivement multijoueurs, les diffusions en direct de vidéo ou audio, etc.

## Le TCP (Transmission Control Protocol)

Le problème avec l'UDP est qu'il n'y a aucune vérification de la bonne réception des paquets ou encore de leur ordre ou de leur intégrité.

Le but du protocole TCP est de garantir l'intégrité des données.

### Transfert fiable

TCP est donc un protocole qui implémente le "transfert fiable", nous allons voir ici en quoi consiste le transfert fiable.

Le transfert fiable est une façon de transférer l'information entre un émetteur et un récepteur de telle sorte à pouvoir palier à des pertes, des duplications, des altérations ou un désordre parmi les informations.

Pour cela, chaque TPDU (Transport Protocol Data Unit) contient un "checksum" permettant de vérifier que l'information n'est pas corrompue → protection contre l'altération.

Et lors de chaque réception d'information, le récepteur doit confirmer la bonne réception, si l'émetteur ne reçoit aucun acquis de bonne réception avant un certain temps (timer), il considère que l'information est perdue et la renvoi → protection contre la perte d'information.

Si l'acquis lui-même est perdu, l'émetteur va renvoyer l'information et le récepteur va réenvoyer son acquis, car ce dernier a déjà reçu l'information → protection contre la duplication et la perte d'acquis.

Chaque acquis et chaque envoi d'information est donc numéroté, il est ainsi possible de savoir pour chaque acquis à quoi il fait référence. Si un acquis est donc envoyé deux fois, l'émetteur pourra savoir à quelle information chaque acquis fait référence et agir en fonction. S'il envoie une information 1, reçoit l'acquis pour 1, puis envoie une information 2 et reçoit de nouveau un acquis pour 1, il ne prendra pas compte du deuxième acquis → protection contre la duplication d'acquis.

Les acquies et les informations étant ainsi numérotées et allant dans un ordre de croissant. Et puis ce que l'émetteur attend toujours d'avoir reçu une confirmation de bonne réception de chaque partie de l'information, ce protocole assure donc que les informations sont reçues dans le bon ordre → protection contre le désordre.

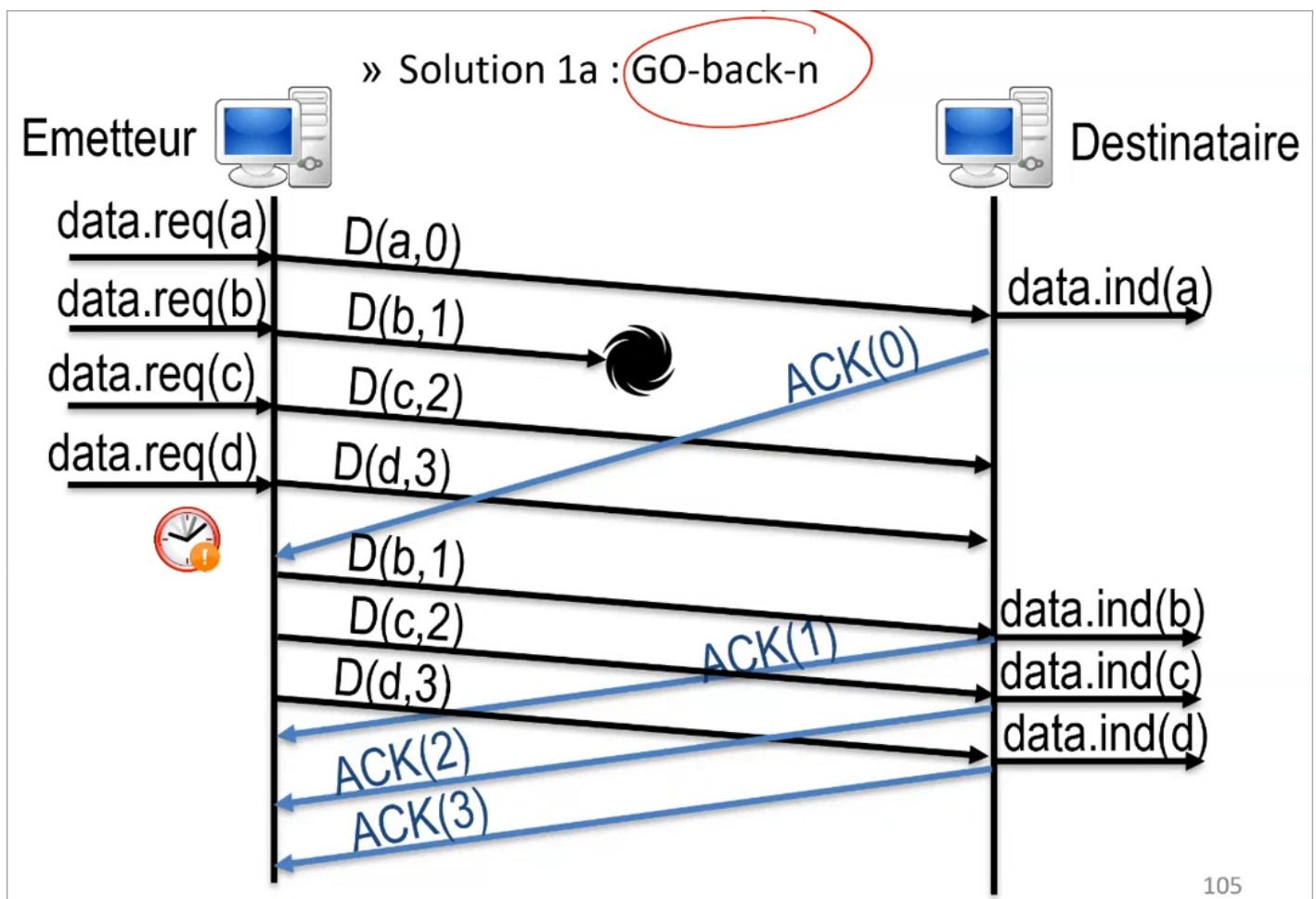
## Fenêtre glissante

Les performances de TCP sont bien plus mauvaises que UDP car si le ping est élevé le round-trip time (temps allé-retour) va être très élevé aussi, cela sera donc très lent de tout transmettre. Pour résoudre ce problème, on peut alors utiliser un système de "fenêtre glissante", on va envoyer plus d'information avant d'attendre un acquies (donc moins d'acquies et pas d'envois de trop de données).

La fenêtre définit une série de numéros de séquences qui peuvent être envoyés sans devoir attendre un acquies. Une fois cette fenêtre épuisée, il faut attendre un acquies (pour toute la fenêtre) pour pouvoir recommencer.

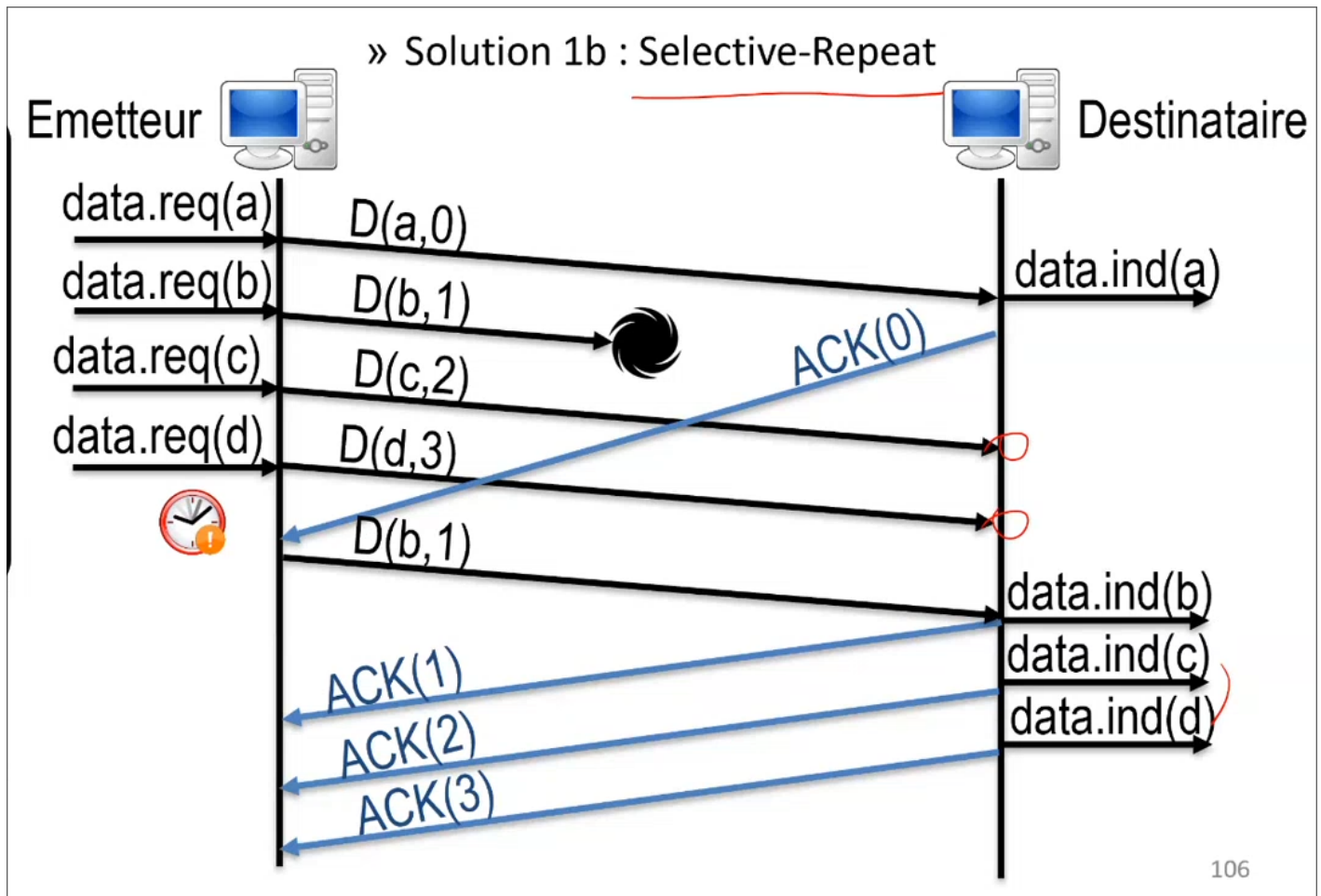
Perte d'information

Lorsqu'une perte d'information survient, il y a plusieurs manières de palier à une perte.

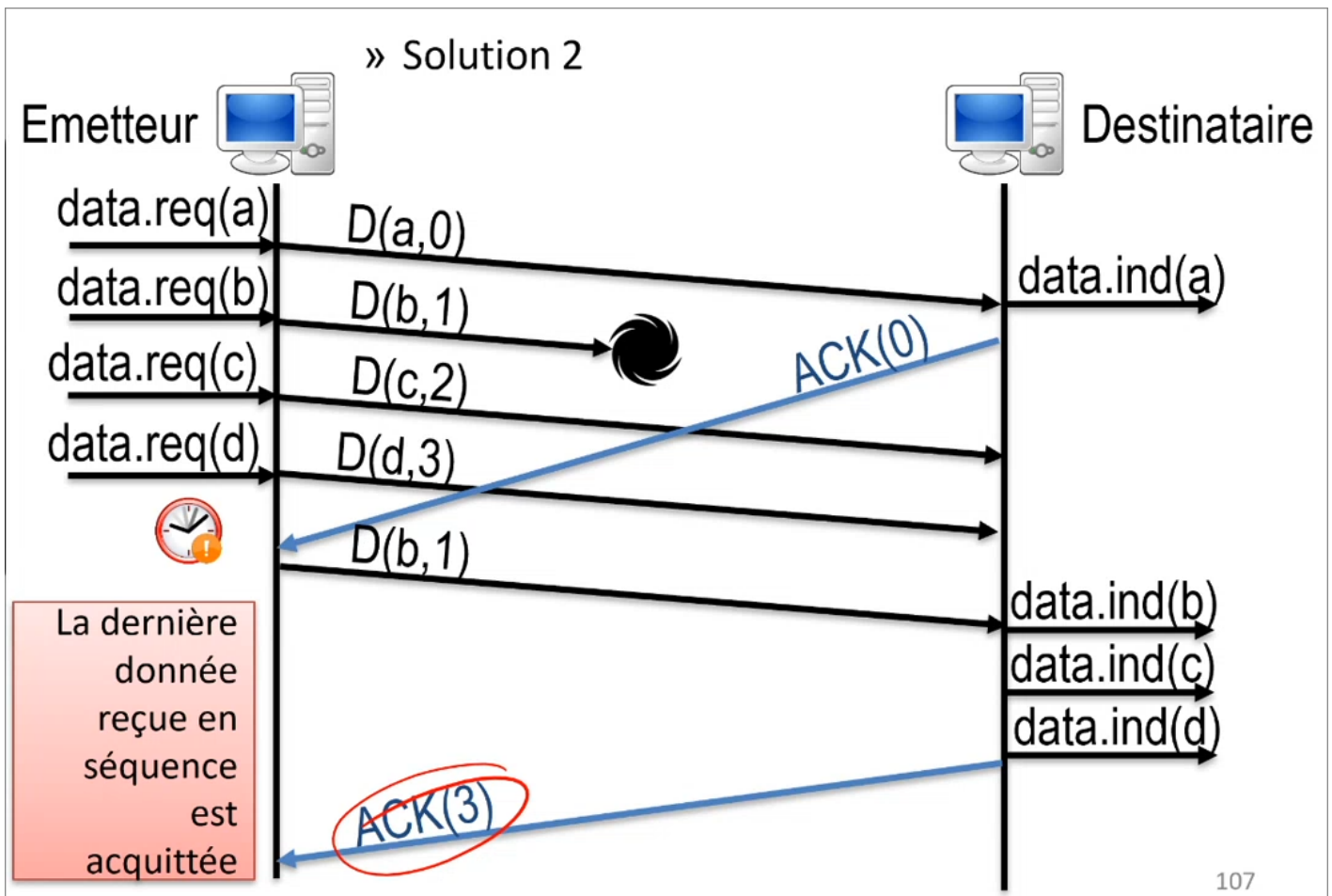


La première, c'est le **go-back-n** dans lequel le destinataire oublie tous les TPDU reçus hors séquence, ainsi s'il doit recevoir 0, 1, 2, 3, mais ne reçoit pas 1, il ne va pas tenir compte de 2 et 3 et va dire à l'émetteur par un acquies : "J'ai seulement reçu le TPDU 0". L'émetteur devra alors renvoyer les TPDU 1, 2 et 3 qui seront alors acquittés par le destinataire.

Cette méthode est avantageuse pour le destinataire, car il n'a pas besoin de retenir les TPDU hors-séquence, mais peu avantageuse pour l'émetteur qui doit tout réenvoyer.



La deuxième méthode est le **Selective Repeat** (plus courante aujourd'hui) qui consiste à garder en mémoire les données hors séquence, si on reprend l'exemple précédent, si on attend de recevoir 0, 1, 2 et 3 et que l'on ne reçoit pas 1, alors on acquitte 0 qui a bien été reçu, l'émetteur envoie alors la donnée suivante, 1. Le destinataire va ensuite acquitter tous les autres paquets reçus (1, 2 et 3) qui ne seront donc pas ré-envoyés.



Pour ne pas avoir à acquitter tout un par un, on peut également acquitter la dernière information reçue en séquence (dans ce cas 3), ce qui équivaut à acquitter 1, 2 et 3 d'un coup, ce qui est donc plus efficace.

#### Capacité de traitement variable

Seulement, la capacité de traitement du destinataire peut varier, c'est pourquoi il va préciser dans ses acquis la taille actuelle de la fenêtre, plus la capacité de traitement du destinataire est grande, plus la fenêtre sera grande, et inversement.

À savoir qu'étant donné que les numéros de séquence sont réutilisés, il est possible d'avoir une duplication d'un acquis avec un certain numéro de séquence avec beaucoup de retard. Cela pourrait poser un problème si par hasard le numéro de séquence actuel est justement celui-là. C'est pourquoi la couche réseau (IP) s'occupe de faire un "timeout" sur les paquets, ainsi les paquets trop anciens sont juste oubliés, ce qui règle donc ce problème.

## Connexion et déconnexion

Pour pouvoir commencer à transférer des données, il faut d'abord établir une connexion pour partager des informations initiales. Pour ce faire, on utilise un **three-way handshake**.

## Le three-way handshake

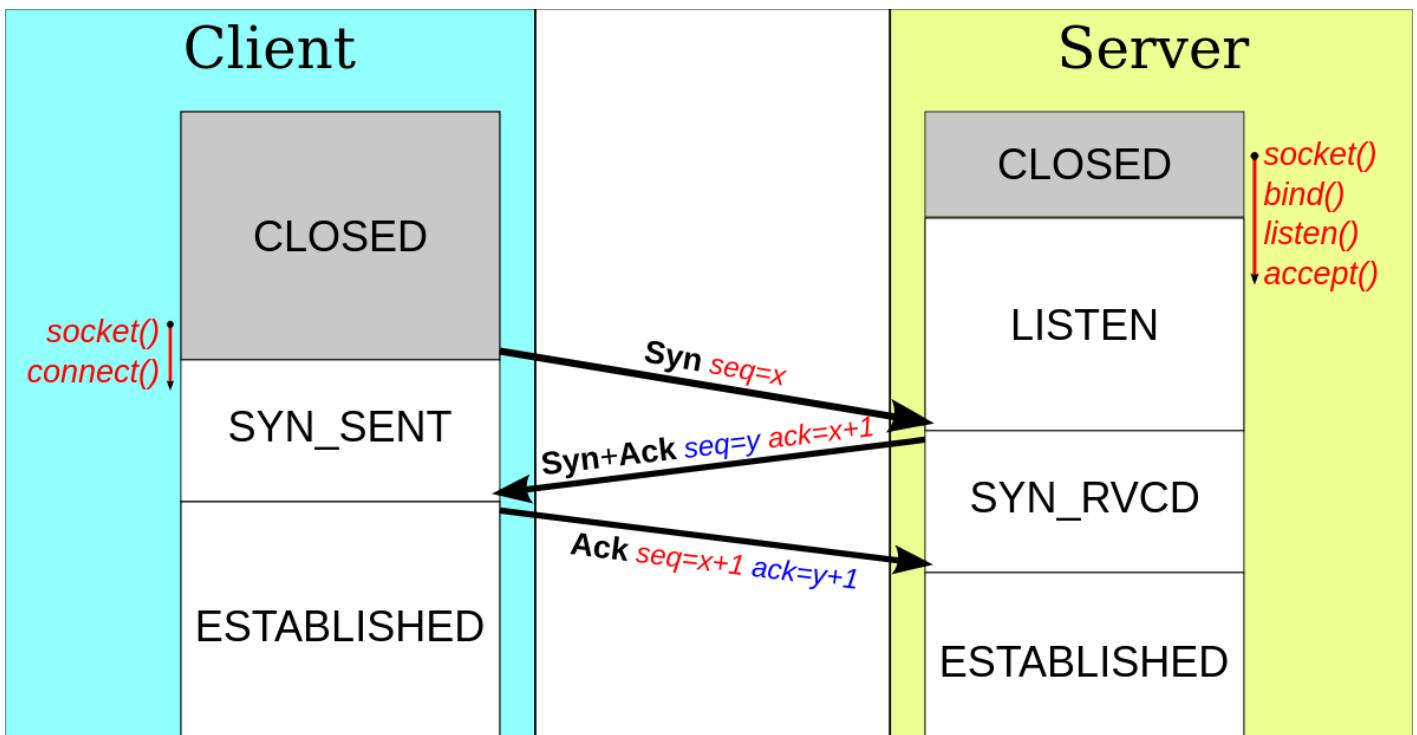
- » L'horloge fournit des numéros de séquence croissants
- » Utiliser ces numéros de séquence lors de la demande de connexion et lors de l'acquit de celle-ci
- » Possibilité de détecter les CR & CA dupliqués



Le client va générer un numéro de séquence (x) et envoyer une demande de connexion au serveur avec ce dernier.

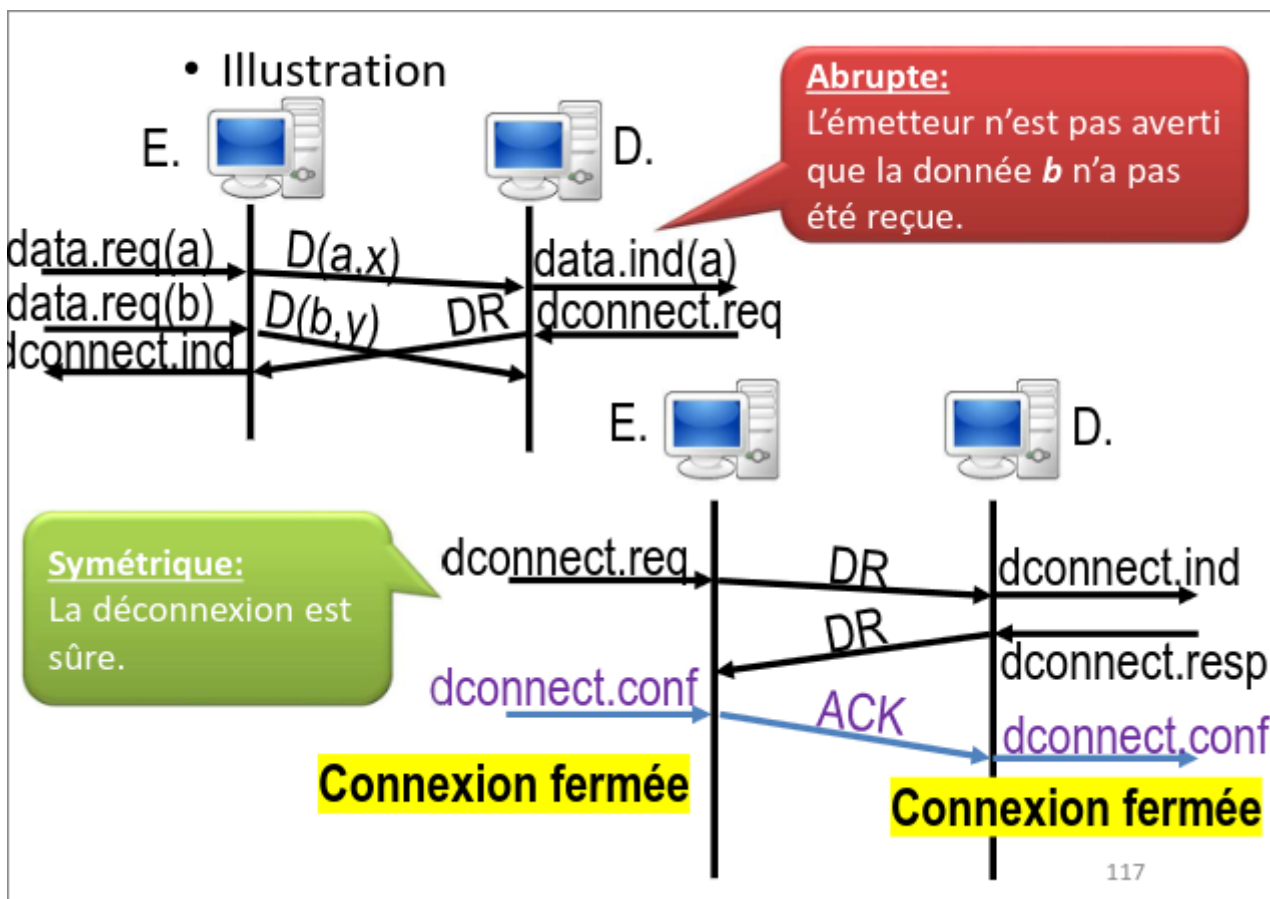
Le serveur va ensuite également générer un numéro de séquence (y) et acquitter la requête, la connexion est alors établie pour le client.

Enfin, le client va acquitter aussi, la connexion est alors établie pour le serveur.



Si une requête est dupliquée, le serveur va envoyer un acquit, mais le client répondra par un `REJECT` pour indiquer que la connexion est refusée, car il n'a pas fait de requête.





Pour ce qui est de la déconnexion, elle peut se faire soit de manière **abrupte**, c'est-à-dire que l'un des deux indique à l'autre "je me casse" et se déconnecte. Le problème, c'est que des données peuvent alors être perdues ou perdre l'information sur la déconnexion.

L'autre méthode est de se déconnecter de manière **symétrique**, autrement dit de manière similaire au three-way handshake. A envoie à B une requête de déconnexion, B envoie à A une requête de déconnexion, A acquitte la requête à B et se déconnecte (et B fait de même).

## Communication bidirectionnelle

Souvent, il arrive que le client et le serveur doivent tous les deux transférer des données, ce qui complique donc un peu les choses.

Ainsi, on peut soit ouvrir deux connexions (une pour client → serveur et une pour serveur → client) mais cela ajoute donc beaucoup de trafic de contrôle et ralentit les choses.

Sinon, on peut utiliser le **piggyback** qui consiste à fusionner les TPDU de contrôle (acquis) et les TPDU de réponse en un seul TPDU ce qui diminue drastiquement donc la quantité de trafic de contrôle.

## Implémentation de TCP

Voir [sur la page suivante](#) pour la description de l'implémentation du transfert fiable dans le protocole TCP.



Revision #3

Created 27 February 2024 16:29:13 by SnowCode

Updated 29 February 2024 14:34:48 by SnowCode