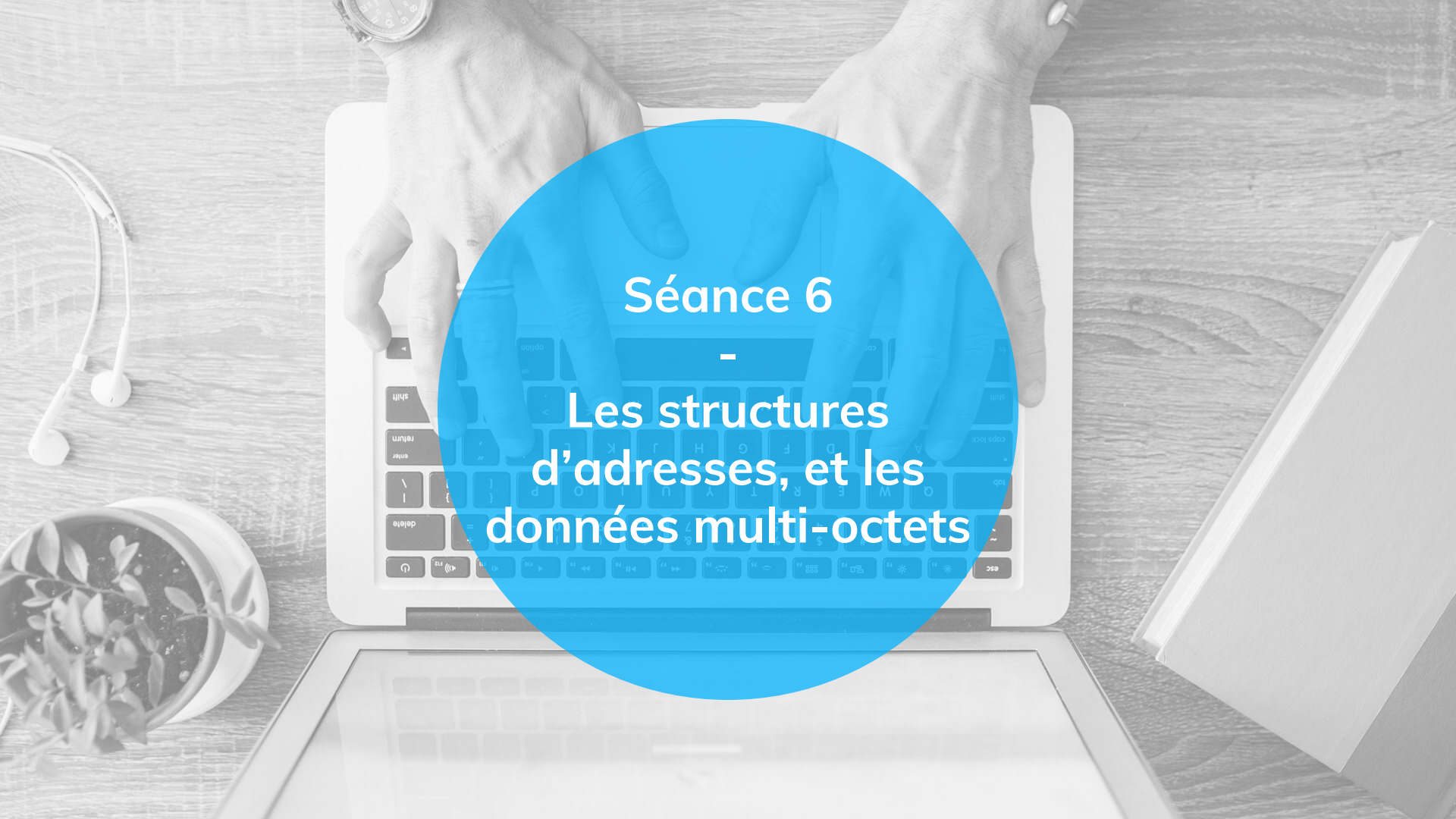


Séance 6

-

Les structures d’adresses, et les données multi-octets

Données multi-octets 1/3

- Avant de commencer le cours sur le réseau, nous allons nous attarder quelque temps sur les structures d'adresses, les données multi-octets, et l'importance de la conversion de données multi-octets pour une utilisation en réseau.
- Il existe en effet deux façons de stocker les données codées sur plus d'un octet , et cela varie en fonction de l'architecture de la machine. On les nomme :
  - Little-Endian
  - Big-Endian
- Little-endian : on lit de droite à gauche les adresses croissantes, mais les bits de poids fort sont à gauche.
- Big-endian : on lit de droite à gauche les adresses croissantes, mais les bits de poids fort sont à droite.
- Petit trick pour savoir si little-endian ou big-endian :
  - `echo -n I | hexdump -o | awk '{ print substr($2,6,1); exit}'`
  - Si 1 : Little-Endian
  - Si 0 : Big-Endian

Données multi-octets 2/3

- Maintenant que nous avons vu le problème des différents stockage des données multi-octets, on peut alors se poser la question du problème de la mise en réseau de ce type de données.
- En effet, une adresse IP par exemple, ou un numéro de port, est codée sur plusieurs octets.
- Pour pouvoir échanger des données multi-octets entre machines, un ordre constant a été défini. On l'appelle l'ordre réseau.
- Quand une donnée multi-octets quitte une machine pour aller sur le réseau, on va la passer de l'ordre host (LE ou BE) à l'ordre réseau, avec une fonction de type hton.
- Inversement, quand une donnée multi-octets arrive du réseau vers une machine, on la passer de l'ordre réseau à l'ordre host, avec une fonction de type ntoh

Données multi-octets 3/3

- On précisera toujours la taille des données concernées avec les suffixes s(short) pour les données sur 2 octets (le port par exemple) et l(long)pour les données sur 4 octets (adresse IP par exemple).
- On a donc les fonction suivantes:
  - De l'ordre hôte à l'ordre réseau :
    - uint32_t htonl(uint32_t hostlong);
    - uint16_t htons(uint16_t hostshort);
  - De l'ordre réseau à l'ordre hôte :
    - uint32_t ntohl(uint32_t netlong);
    - uint16_t ntohs(uint16_t netshort);
- Chaque fonction convertit la donnée passée en paramètre et la retourne dans l'ordre demandé. Si la valeur était déjà dans l'ordre voulu, alors rien ne se passe.

Conversion d’adresses

- Le système manipule les adresses comme un entier de 32 bits, alors que le développeur quant à lui utilise leur forme décimale pointé (exemple : 127.0.0.1).
- On doit donc utiliser des fonctions pour passer d’une forme à une autre :
- int inet_pton(int af, const char *src, void *dst);
    - af est la famille d'adresses (AF_INET pour IPv4).
    - src est la chaîne contenant l'adresse IP.
    - dst contiendra le résultat de la conversion.
    - L'adresse convertie sera stockée en ordre réseau. (pton = presentation to numeric)
- Pour passer une adresse de sa forme numérique à sa forme décimale pointée, on utilise :
- char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
    - af est la famille d'adresses (AF_INET pour IPv4).
    - src est l'adresse sous sa forme numérique.
    - dst contiendra la chaîne résultant de la conversion.
    - size est la taille de la chaîne dst.
    - (ntop = numeric to presentation)

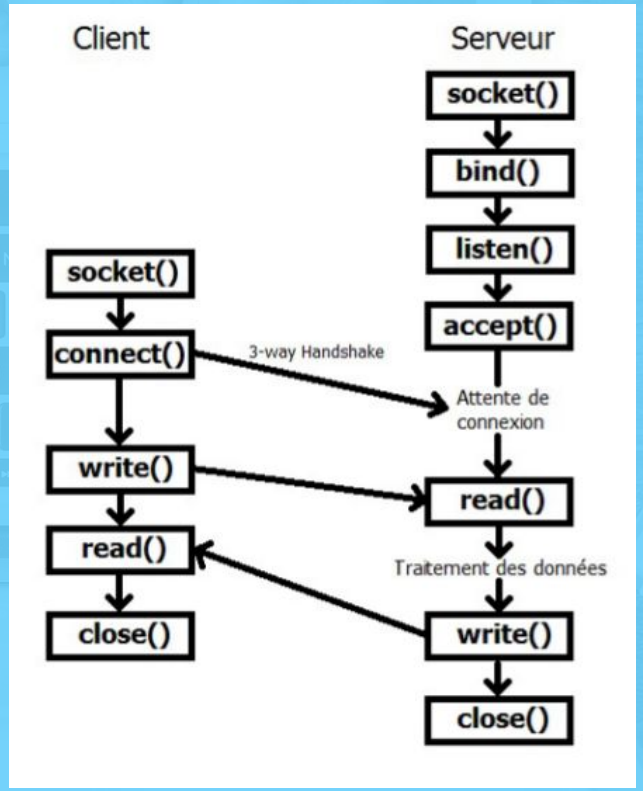
Séance 7

-

Les sockets TCP

Le serveur mono-processus (non-concurrentiel)

- Lors d'une connexion TCP entre un client et un serveur, les fonctions sont appelées dans un ordre bien précis :



Les structures d’adresse

- 1 – Ipv4 :
  - struct sockaddr_in
    {
        short sin_family;
        u_int8 sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
    };
- 2 – Générique :
  - struct sockaddr
    {
        short sa_family;
        char sa_data[14];
    }

Fonction socket

Sockets TCP : socket()

- Les sockets se manipulent via des file descriptors ! (donc toute les fonctions qu'on a vu au début de ce cours fonctionnent encore et toujours !).
- Pour créer une socket active (qui peut initier une connexion), on utilise la fonction socket()

- Prototype :
  - int socket(int family, int type, int protocol);
- Retours :
  - Si succès, retourne le descripteur de la socket active
  - Sinon, -1
- Fonctionnement :
  - family : famille d'adresse de la socket : AF_INET pour IPv4.
  - type : type de socket :
    - SOCK_STREAM : socket TCP.
    - SOCK_DGRAM : socket UDP.
    - SOCK_RAW : socket RAW.
  - protocole : protocole support de la socket. On laisse ce paramètre à 0 : le noyau détermine le protocole en fonction des 2 paramètres précédents

Fonction bind

Sockets TCP : bind()

- Prototype :
  - int bind(int sockfd, struct sockaddr *myaddr, socklen_t addrlen);
- Valeurs de retour :
  - Si succès, retourne 0,
  - -1 sinon.
- Fonctionnement :
  - sockfd : descripteur de la socket à lier.
  - myaddr : structure d'adresse IPv4 contenant l'adresse IP et le port à lier à la socket (en ordre réseau). Cet argument sera transtypé en structure d'adresse générique.
  - addrlen : taille de la structure d'adresse passée en deuxième paramètre.
- Cette fonction n'est utilisée que dans le serveur. En effet, le client doit préalablement savoir où se connecter, et définit donc ainsi un couple IP/PORT qu'il connaît.
- Si le serveur veut écouter sur toutes les interfaces, on utilisera l'adresse spéciale (Wildcard) INADDR_ANY
- Le client quant à lui, va indiquer dans connect le couple, et n'aura pas besoin de bind (le noyau va le gérer seul)

Fonction listen

Sockets TCP : listen()

- Prototype :
  - int listen(int sockfd, int nbconn);
- Valeurs de retour :
  - Si succès, retourne 0,
  - -1 sinon.
- Fonctionnement :
  - sockfd : descripteur de la socket à rendre passive.
  - nbconn : nombre de connexions établies ou en attente d'établissement maximum pour la socket.
- Transforme une socket active (capable d’initier une connexion) en socket passive (capable d’accepter une connexion)

Fonction accept

Sockets TCP : accept()

- Prototype :
  - int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *clilen);
- Valeurs de retour :
  - Si succès, retourne le descripteur de la socket connectée,
  - -1 sinon.
- Fonctionnement :
  - sockfd : descripteur de la socket passive du serveur.
  - cliaddr : si différent de NULL, contiendra les coordonnées (adresse IP et port) du client qui s'est connecté, en ordre réseau. L'argument devra être transtypé en structure d'adresse générique.
  - clilen : si non NULL, prendra la taille de la structure du client connecté.
- Place le serveur en attente de connexion.
- Tant qu'une connexion n'a pas été établie complètement (3-way handshake de TCP), le processus serveur est endormi. Il sera réveillé à la fin de l'établissement de la connexion.

Fonction connect

Sockets TCP : connect()

- Prototype :
  - int connect(int sockfd, struct sockaddr *servaddr, socklen_t servlen);
- Valeurs de retour :
  - Si succès, retourne 0,
  - -1 sinon.
- Fonctionnement :
  - sockfd : descripteur de la socket active du client.
  - servaddr : structure d'adresse IPv4 contenant l'adresse IP et le port du serveur auquel le client veut se connecter (en ordre réseau). Cet argument sera transtypé en structure d'adresse générique.
  - servlen : taille de la structure d'adresse passée en deuxième paramètre.
- Initie une connexion du client vers le serveur.
- Pas besoin de bind pour le client : c'est le noyau qui décidera des 2 valeurs IP et PORT:
  - l'adresse IP sera choisie en fonction du routage
  - le port sera choisi aléatoirement parmi les ports non réservés (supérieurs à 1024) et sera ouvert le temps de la connexion (port éphémère)

Fonction read
Fonction write
Fonction close

Sockets TCP : read(), write(), close()

- read et write :
- Pour lire ou écrire dans une socket, on utilise les appels-système read et write vu au début du cours.
    - read sera bloquant : s'il n'y a aucune donnée à lire sur la socket, mais que la connexion est établie et qu'on appelle read, le processus est mis en attente.
    - Si la valeur de retour de read est égale à 0, cela signifie que l'autre côté a clos la connexion
- close :
- Nous utilisons la même fonction close que vue au début du cours.
- L'appel à close provoque la fin de la connexion et le début de la phase de déconnexion TCP si et seulement si plus aucun processus ne référence la socket comme ouverte.
- A chaque processus qui utilise une même socket, un compteur de référence est incrémenté pour la socket. Chaque appel à close décrémente ce cce compteur : la socket ne sera effectivement fermée que lorsque ce compteur atteint la valeur 0

TP-10
Fonctions réseau

Les Sockets TCP - TP-10

- **Exercice 1 :**
  - **Développer le programme correspondant au schéma concernant le serveur mono-processus (non-concurrentiel)**

Serveur mono-processus ou multi-processus ?

- Le serveur que nous avons vu jusqu'à présent est un serveur basique : il ne peut traiter qu'une connexion à la fois. Une fois un client connecté, aucun autre ne peut s'y connecter.
- Pour traiter plusieurs requêtes simultanées, il faut créer un serveur concurrent. On peut alors développer un serveur multi-processus.
- Le principe est le suivant :
  - le serveur se met en attente de connexion.
  - Une fois un client connecté, le serveur crée un processus, qu'on appellera serveur fils.
  - Celui-ci est chargé de traiter la connexion pendant que le serveur père se remet en attente de connexion.
- A chaque nouvelle connexion, un processus serveur fils est créé.
- Le serveur fils hérite, lors du fork, des fd et sockets du père.
- Or, le fils n'utilisera jamais le descripteur de socket passive et le père n'utilisera plus le descripteur de socket connectée. Il faut donc que chacun des processus ferme le descripteur qu'il n'utilise plus.

Thanks!

Questions?

CREDITS

Remerciements aux personnes ayant fourni ce template gratuitement sous common license :

- Simple line icons by Mirko Monti
- E-commerce icons by Virgil Pana
- Streamline iconset by Webalys
- Presentation template by SlidesCarnival
- Photographs by Death to the Stock Photo (license)