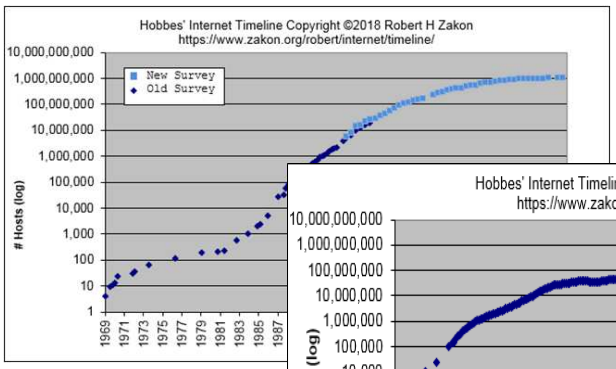
Programmation Réseaux

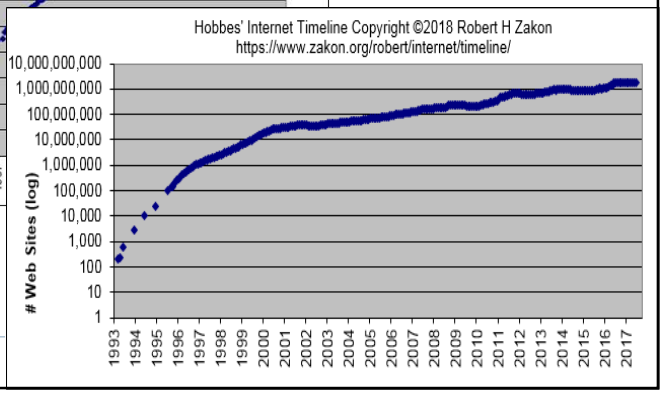
www.elabedabir.weebly.com

Internet : l'Histoire

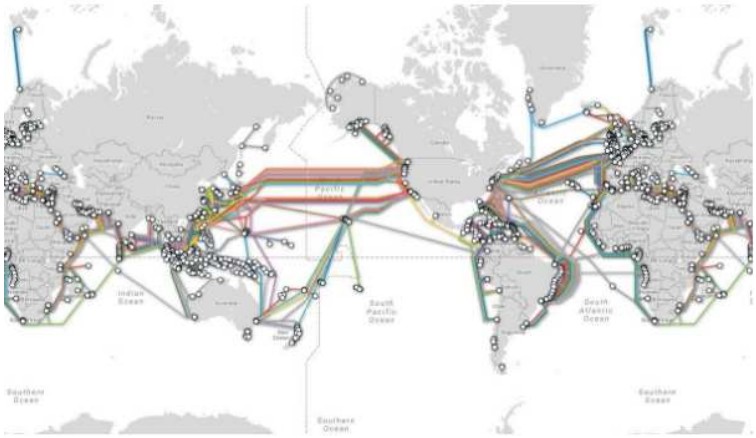
Internet Hosts



WWW Growth



Le réseau physique mondial



- https://www.weforum.org/agenda/2016/11/this-map-shows-how-undersea-cables-move-internet-traffic-around-the-world/

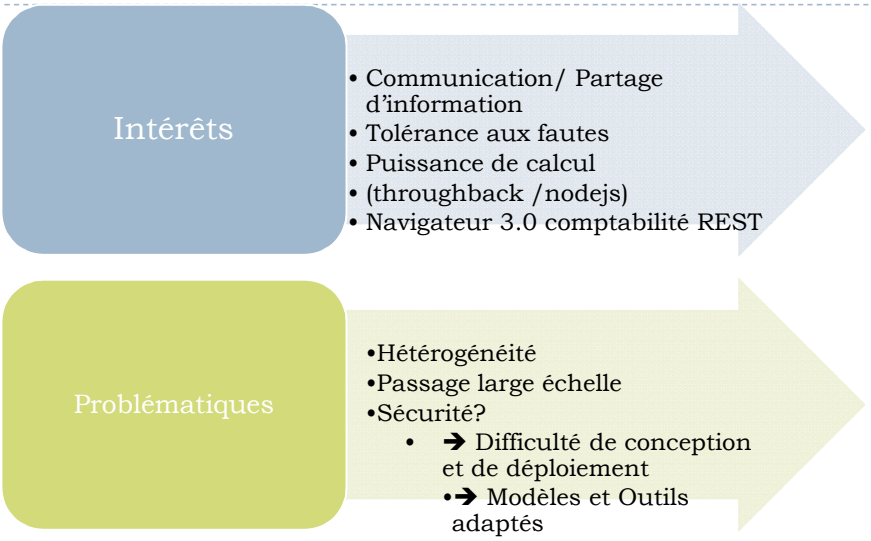
AS-Peering

- Le réseau Internet est composé de différents réseaux indépendants appelés AS (*Autonomous System)*
- Un AS est une région de l'Internet gérée par une unité unique
  - Exemple:
    AS25 - UCB - University of California at Berkeley
    AS15169 - GOOGLE - Google Inc.
    AS5438 - ATI-TN Agence Tunisienne d'Internet
- Interconnexion
  La connexion entre les différents AS se fait au niveau de ***Internet Exchange Point.*** *Les échanges sont soumis à des* accords de *peering (homologation)*
- Routage Intra-domain (RIP, OSPF, IGRP, IS-IS)/ Inter-domain (BGP)

Quelques organismes de l'Internet

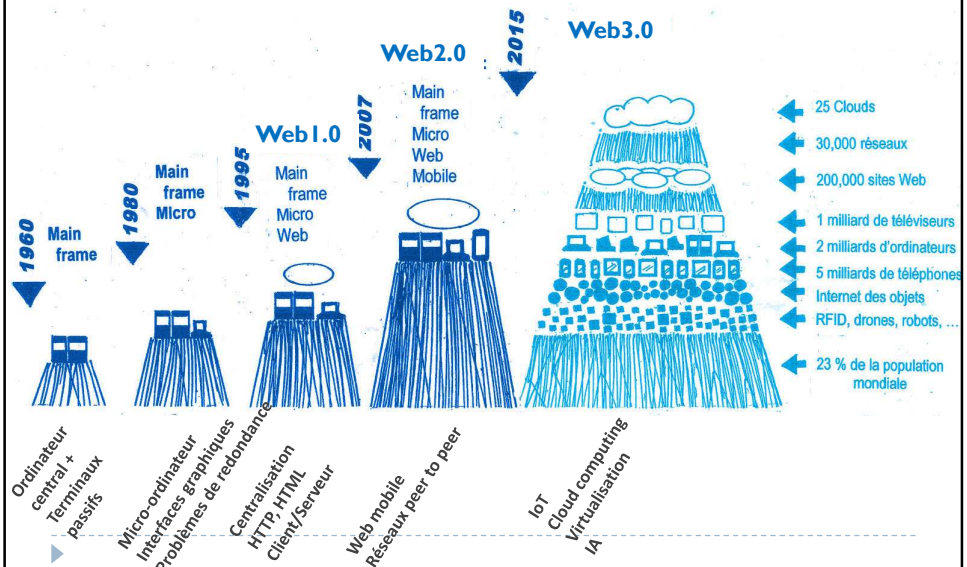
- ISOC (*Inernet Society*) O.N.G américaine
- ICANN (*Internet Corporation for Assigned Names and Numbers*) ONG américaine
- IANA (*Internet Assigned Numbers Authority*) fait partie de l'ICANN
- RIR (*Regional Internet Registery*), *LIR* (*Local Internet Registery*) ;
- IETF (*Internet Engineering Task Force*)
- *IRTF* (*Internet Research Task Force*)
- W3C (*World Wide Web Consortium*)

La répartition



Le besoin de répartition

De la naissance de l'Internet jusqu'à d'aujourd'hui internet➔ web 3.0



Projets de Programmation Réseau

1. Application de transferts de fichiers
2. Gestion d'une machine distante
3. Sniffer réseau avec rawsockets
4. Application de chat avec les sockets
5. Socket avec une Applet (*nouveau Java Web Start)*
6. Socket en Python
7. RPC en C
8. RMI de java
9. Client/ serveur de messagerie javaMail
10. News avec MQTT - Mosquito
11. Application de chat /jeu en ligne avec XMPP - Jabber
12. Communication par messages :AMQP avec RabbitMQ
13. Smart Contract sur blockchain
14. Conteneurs (Docker…)
15. Les micro-services
16. Application de chat avec NodeJs

Le paradigme Client/Serveur

- Internet fonctionne principalement sur un modèle de client / Serveur.
- Les calculs peuvent être fait côté serveur ou côté client (Client lourd/léger)
- Suivant les contraintes d'utilisation ou contraintes techniques on a des différents types
  - ✓ Architecture 1-tiers
  - ✓ Architecture 2-tiers
  - ✓ Architecture 3-tiers
  - ✓ Architecture n-tiers

Découpage d'une application

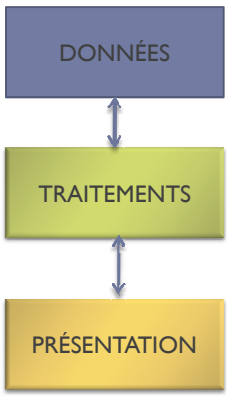
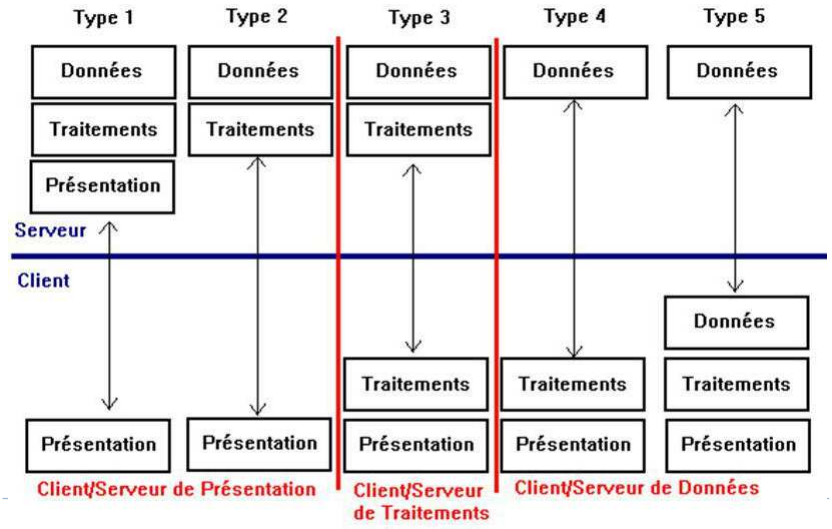
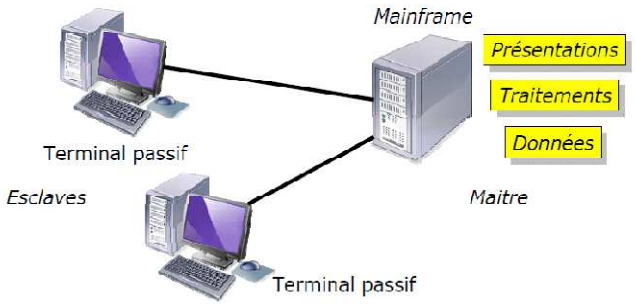


Schéma de Gartner Group pour la distribution du C/S



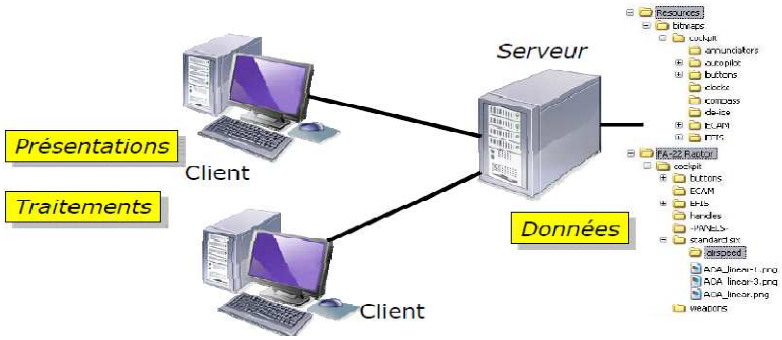
L'architecture 1 tiers

- ✓ Les 3 couches applicatives s'exécutent sur la même machine
- ✓ On parle d'informatique centralisée :
- ✓ Contexte multi-utilisateurs dans le cadre d'un site central (mainframe)



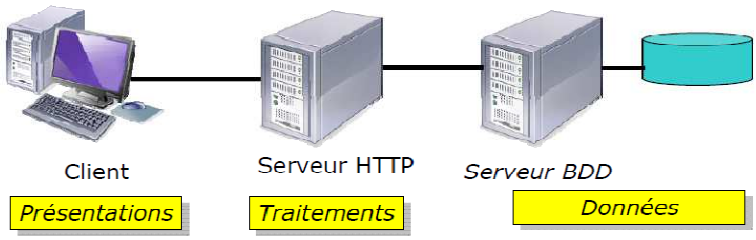
L'architecture 2 tiers

- ✓ Présentation et traitements sont sur le client
- ✓ Les données sont sur le serveur
- ✓ Contexte multi-utilisateurs avec accès aux données centralisées



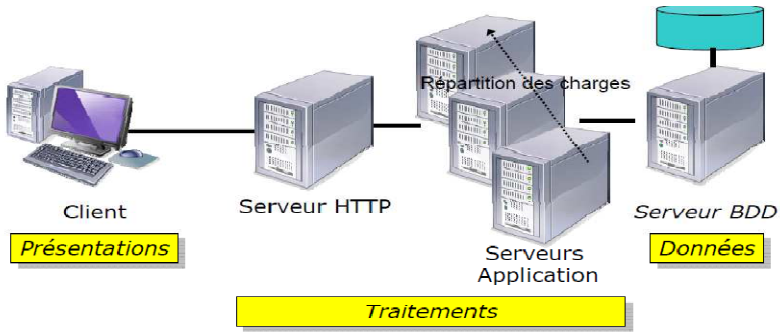
L'architecture 3 tiers

- ✓ La présentation est sur le client
- ✓ Les traitements sont pris par un serveur intermédiaire
- ✓ Les données sont sur un serveur de données



L'architecture N tiers

- ✓ La présentation est sur le client
- ✓ Les traitements sont pris par un serveur intermédiaire
- ✓ Les données sont sur un serveur de données

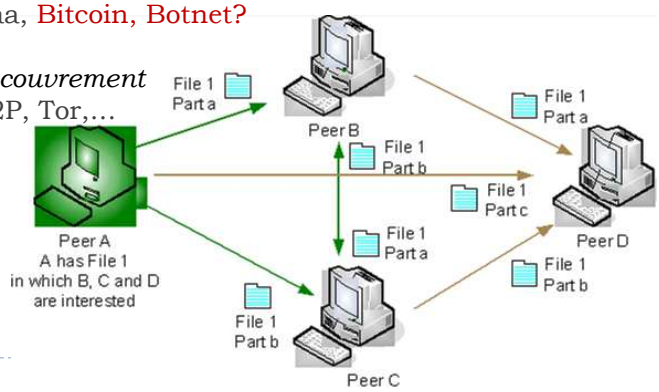


Serveurs … ou pas?
L'architecture peer to peer

- ✓ Chaque nœud du réseau est libre de partager ses ressources.
- ✓ Un nœud peut jouer le rôle de Serveur / Client

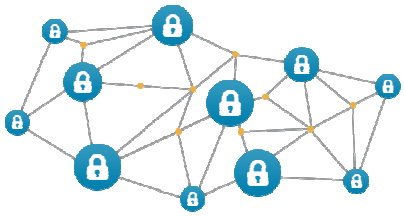
Applications
BitTorrent, eMule, Napster, eDonkey
Kademlia, KaZaa, Bitcoin, Botnet?

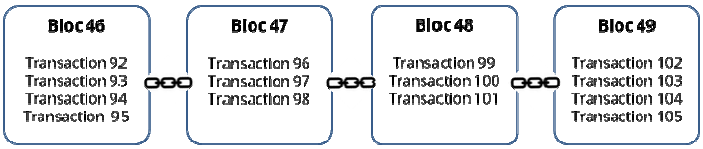
Protocoles de recouvrement
Mnet, Chord, I2P, Tor,…



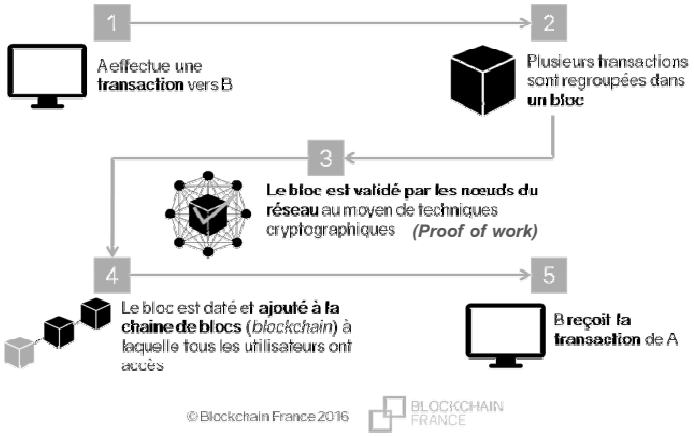
Architecture BlockChain

- ✓ Architecture décentralisée peer to peer
- ✓ Les transactions effectuées entre les utilisateurs du réseau sont regroupées par blocs.
- ✓ Chaque bloc est validé par les nœuds du réseau appelés les "mineurs",



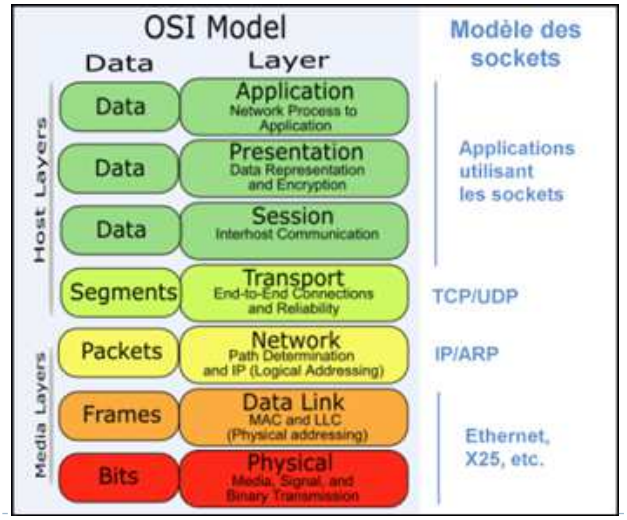


Architecture BlockChain (2)

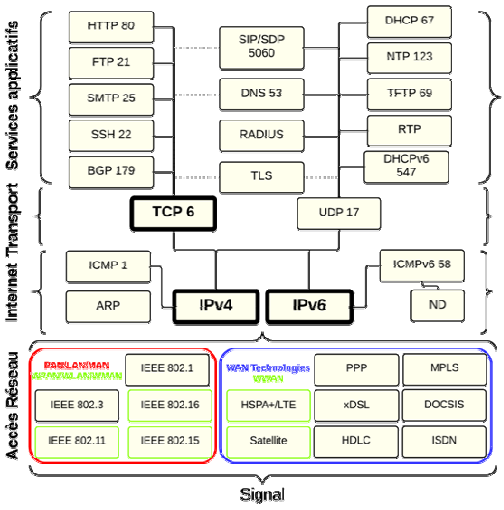


- ✓ Blockchain publique/privée
- ✓ Exemples de protocoles :
  - Bitcoin (BTC), Ethereum (Ether), Ripple (XRP), …

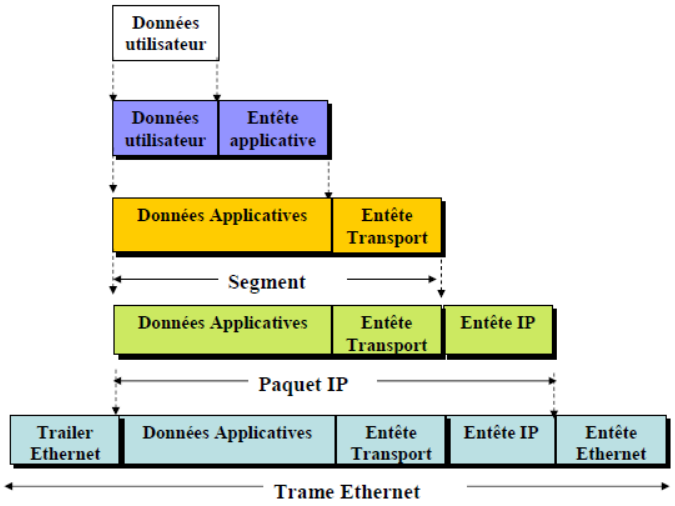
Les modèle en couches



Le modèle TCP/IP



L'encapsulation TCP/IP



Principe «classique» de Client/Serveur

- Repose sur une communication entre les processus applicatifs :
  - un processus client
  - un processus serveur
- Les processus ne sont pas identiques mais forment un système coopératif se traduisant par un échange de données

  Le client reçoit les résultats finaux délivrés par le serveur
- Le client initie l'échange
- Le serveur est à l'écoute d'une requête cliente éventuelle
- La communication peut être synchrone/asynchrone

Les protocoles Applicatifs

- Le protocole applicatif définit
  - Le format des messages échangés entre émetteur et récepteur (textuel, binaire, MIME, Unicode, UTF-8 …)
  - Les types de messages : requête/ réponse / informationnel …
  - L'ordre d'envoi des messages
- Ne pas confondre protocole et application
  - Une application peut supporter plusieurs protocoles (ex : logiciel de messagerie supportant POP, IMAP et SMTP)
  - Application de transfert de fichiers supportant FTP, TFTP, SFTP,…
  - Serveur Web supportant HTML, MQTT, CoAP…

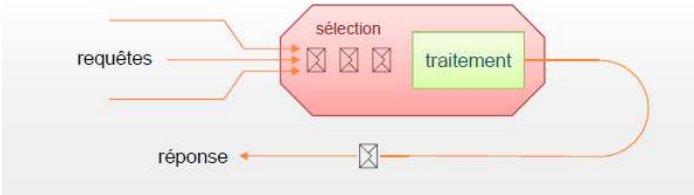
Gestion des Processus

- Mode de gestion des requêtes
  - Itératif : le processus serveur traite les requêtes les unes après les autres
  - Concurrent basé sur
    - Parallélisme réel
      - ➔ Système multiprocesseurs par exemple
    - Pseudo-parallélisme
      - ➔ Schéma veilleur-exécutants (*Daemon*)
    - La concurrence peut prendre plusieurs formes:
      - Plusieurs processus (un espace mémoire par processus)
      - Plusieurs processus légers (*threads)* dans le même espace mémoire

Gestion des Processus dans le serveur

- Processus serveur unique

```
while (true) {
    Receive (client_id, message)
    Extract (message, service_id, params)
    Do_service[service_id] (params, results)
    Send (client_id, results)
}
```



Gestion des Processus dans le serveur (2)

- Schéma veilleur-exécutants

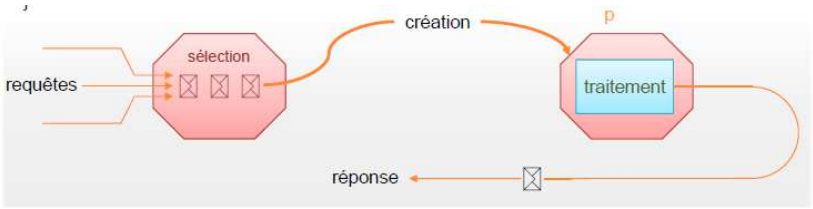
Processus veilleur

```
while (true) {
    Receive (client_id, message)
    Extract (message, service_id, params)
    p = create_thread (client_id, service_id, params)
}
```

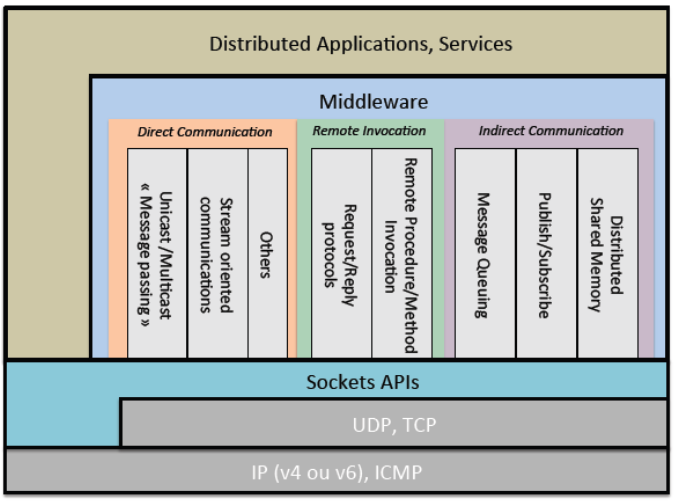
Processus exécutant

Programme de p

```
Do service[service id] params, results)
Send (client_id, results)
exit
```



Mise en œuvre du modèle Client/Serveur
Les modèles de communication



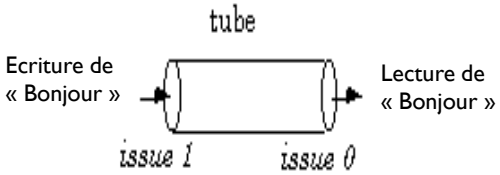
Les tubes nommés (*FIFO*)

- Named Pipes
  - Ils font partie de la norme POSIX sous le nom de fifo
  - Basés sur une communication par fichiers
    - Ils ont une référence dans le système de fichiers
    - Un processus connaissant cette référence peut récupérer un descripteur via la fonction ***open***
    - Autorise la communication entre plusieurs processus (***unidirectionnel***)
- Fonction de création d'un tube nommé

  int mkfifo(char *nom, mode_t mode);

  Le paramètre nom indique le nom du tube à créer et le paramètre mode, ses droits

Les tubes nommés (*2*)




```
int main ()
{char s[30] = « Bonjour »;
int fd;

if (mkfifo(« fifo », 0666)==-1)
   perror (« mkfifo »);
fd=open(« fifo », WRONLY);
…write (fd, s, strlen(s)));
…
```

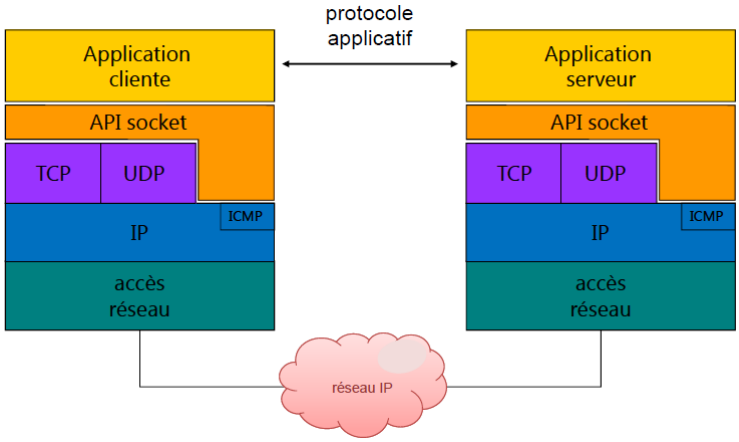
```
int main ()
{char s[30];
Int fd;

if (mkfifo(« fifo », 0666)==-1)
  perror (« mkfifo »);
fd=open(« fifo », RDONLY);
…read(fd, s, 30);
…
```

Les Sockets

- API (*Application Program Interface) socket*
  - Mécanisme d'interface de programmation
    - Permet aux programmes d'échanger des données
    - Les applications client/serveur ne voient les couches de communication qu'à travers l'API socket (abstraction)
    - N'implique pas forcément une communication par le réseau
- L'API socket s'approche de l'API fichier d'Unix
  - Est concurrent aux *Named pipes (tubes nommés)*
- Développée à l'origine dans Unix BSD (*Berkeley Software Distribution)*

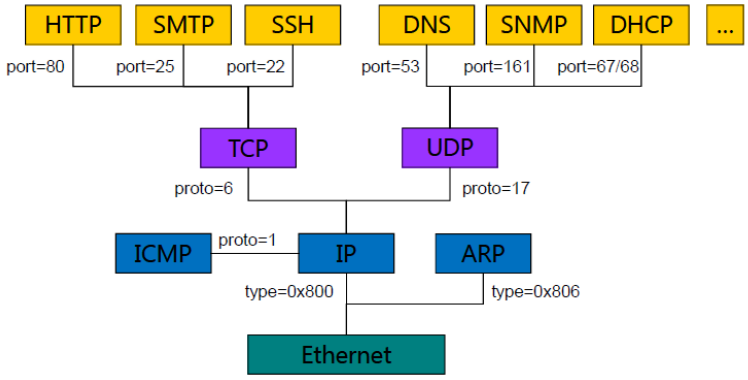
L'API Socket



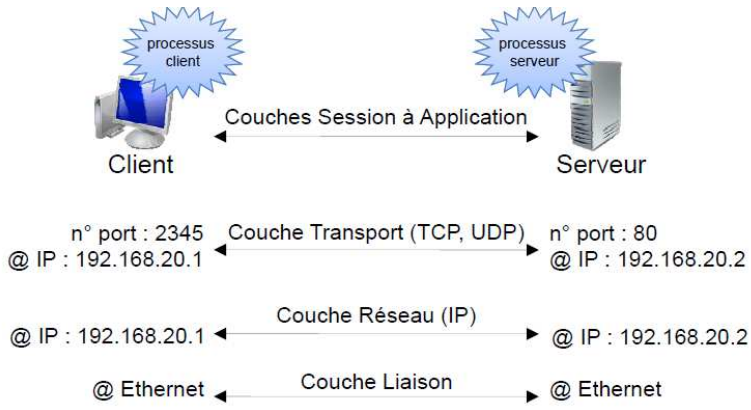
Notion de port

- Un service rendu par un programme serveur sur une machine est accessible par un port
  - Un port est identifié par un entier (16 bits)
    - De 0 à 1023
      - ports **reconnus ou réservés**
      - sont assignés par l'IANA (*Internet Assigned Numbers Authority)*
      - donnent accès aux services standard : courrier (SMTP port 25), serveur web (HTTP port 80) …
    - > 1024
      - ports « **utilisateurs** » **disponibles pour placer un** service applicatif quelconque
- Un service est souvent connu par un nom (FTP, …)
- La correspondance entre nom et numéro de port est donnée par le fichier /etc/services

Identification des protocoles



Notion de port (2)



Les sockets

- Avec les protocoles UDP et TCP, une connexion est entièrement définie sur chaque machine par:
  - Le type de protocole (UDP ou TCP)
  - L'adresse IP
  - Le numéro de port associé au processus
    - Port serveur : port local sur lequel les connexions sont attendues
    - Port client : alloué dynamiquement par le système

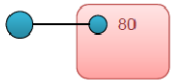
Identification de connexions

La combinaison de 2 sockets définit une connexion TCP ou un échange UDP

- Exemple: 130.190.5.1 -23 et 147.171.150.2 -1094
  - Connexion entre un processus client qui a pris le numéro 1094 sur la machine 147.171.150.2 et le démon *telnetd* sur la machine 130.90.5.1
  - Un utilisateur sur 147.171.150.2 a fait un *telnet 130.190.5.1*
  - C'est ce qu'on peut voir avec la commande Unix *netstat -a*

Mode de fonctionnement

1. Le serveur crée une « Socket Serveur » (associée à un port) et se met en attente.



2. Le client se connecte à la Socket Serveur
   - 2 sokets sont alors créées
     - 1 « client » côté client
     - 1 « socket service client » côté serveur

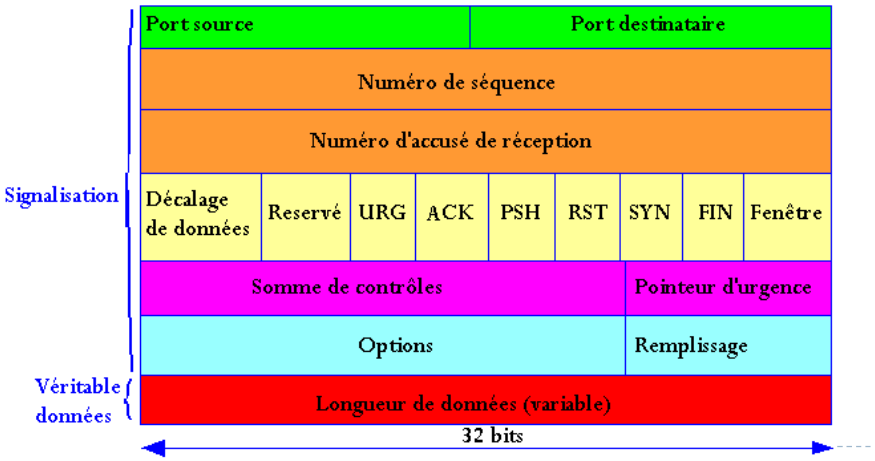
3. Le serveur et le client communiquent à travers la socket créée
   - L'interface est celle des fichiers (*Read, Write)*
   - *La Socket Serveur peut accepter des nouvelles connexions*

Mode Connecté/ Non Connecté

- Deux réalisations possibles
  - Mode connecté (protocole TCP)
  - Mode non connecté (protocole UDP)
- Mode connecté
  - 3 phases: Etablissement de connexion /Transfert/ Libération de connexion
  - Le serveur préserve son état entre deux requêtes (***Stateful protocol***)
  - Garanties de TCP : ordre, contrôle de flux, fiabilité
  - Adapté aux transferts continus (trafics par flots/élastiques/en streaming)
  - Ne tourne pas aux routeurs !
    - Seulement aux extrémités

TCP (RFC793)

- Une entête multiple de 32 octets
  - Format assez complexe…

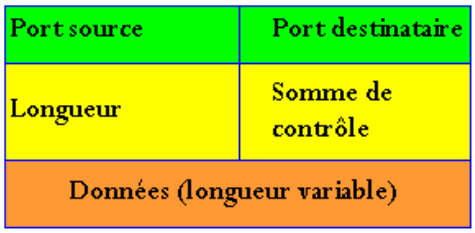


Mode Connecté/ Non Connecté (2)

- Mode non connecté
  - Les requêtes successives sont indépendantes
  - Pas de préservation de l'état entre les requêtes (***Stateless protocol***)
  - Le client doit indiquer les adresses source/destination à chaque requête
  - Pas de garanties particulières (UDP)
    - Gestion de toutes les erreurs à la main : il faut réécrire la couche transport !!!
  - Adapté :
    - Aux transferts brefs (ponctuels/ petit nombre de messages)
    - À la diffusion
    - Au trafic temps-réel

UDP (RFC768)

- Entête de 8 octets
- Source Port: numéro de port
  - Optionnel, identifie un port pour la réponse
- Destination Port: numéro de port de destination
- Length: taille de l'entête et des données
  - Unité = octet
  - Taille max = 64 Koctets
- Cheksum : fonction de l'entête et des données
  - Optionnel
  - C'est la seule garantie sur la validité des données qui arrivent à destination



Primitives de service de la couche Transport

- The primitives for a simple transport service:

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

- Example: Socket primitives for TCP:

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Sockets et Association

- Données necessaires pour identifier un canal client-serveur
  → `BIND` (asociation)

{protocole, adresse locale, processus local, adresse distante, processus distant}

- Moyen d'association dans la machine locale:

{protocole, adresse locale, processus local}

- Moyen d'association dans la machine distante:

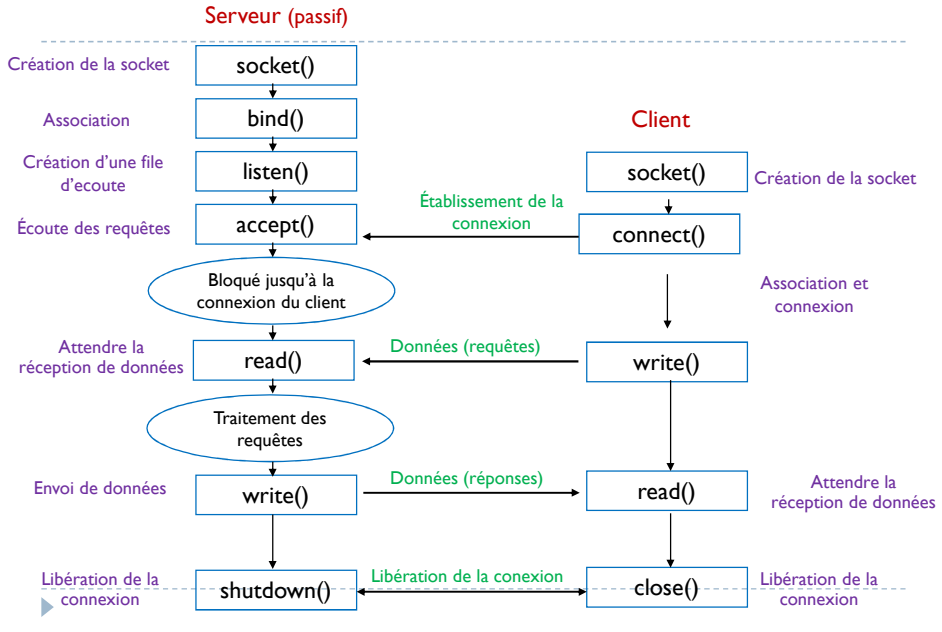
{protocole, adresse distante, processus distant}

Exemple: #ftp 193.147.56.8

→ IP distant: 193.147.56.8
→ port distant: 21
→ IP local: celle du client (exemple: 193.147.56.13)
→ port local: 1500

{TCP, 193.147.56.8, 21, 193.147.56.13,21} ➔
{TCP, 193.147.56.8, 21} → socket dans le serveur
{TCP, 193.147.56.13,1500} → socket dans le client

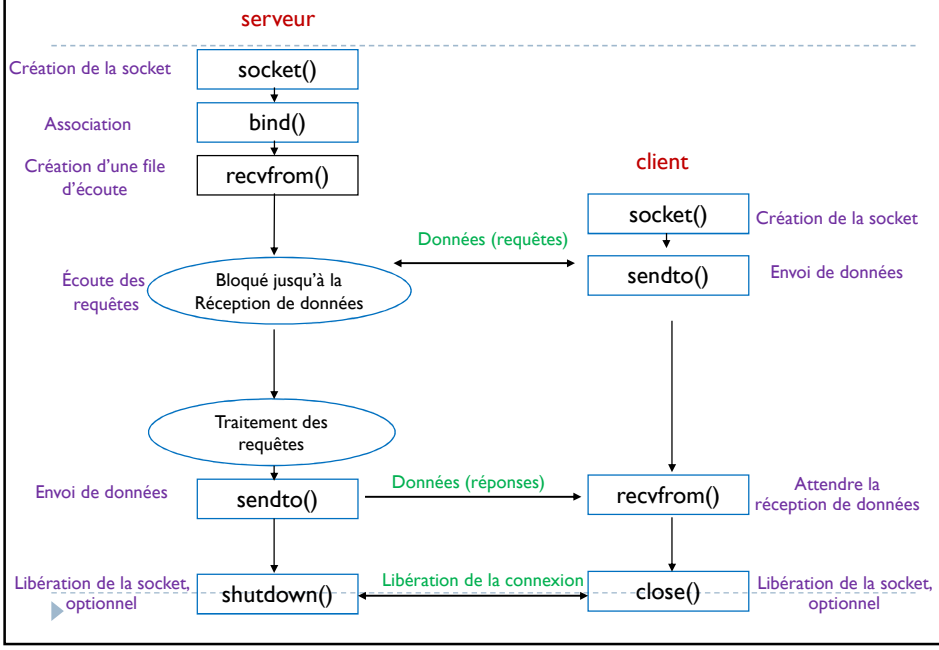
Schéma de communication en mode connecté



Enchaînement des procédures en mode connecté

1. L'appel système ***socket*** permet d'indiquer au système que l'application désire communiquer avec un autre programme. Cet appel système retourne un identificateur , qui sera utilisé dans tous les autres appels systèmes faisant référence à cette socket
2. L'appel ***bind*** permet à l'application de fournir à l'OS les paramètres nécessaires au fonctionnement de la socket. Pour un serveur, il s'agit du port sur lequel il attendra les demandes de connexion
3. L'appel ***listen*** permet de placer la socket et le protocole associé dans un état d'attente de requêtes de connexion
4. L'appel système ***accept*** est bloquant. Il place le programme en attente effective de requête de connexion. Quand une connexion est établie par un client, l'appel système ***accept*** se débloque et retourne un identificateur de socket sur lequel des données pourront être échangées
5. Les appels ***read*** et ***write*** permettent l'échange de données
   ** *Cf* recv / send
6. L'appel système ***close*** permet de fermer localement la socket

Schéma de communication en mode non connecté



Domaines de socket

- Indique sous quel réseau la communication va se réaliser
  - ⇨ adressage à employer
- Structure d'adresses d'un socket <sys/socket.h>

```
struct sockaddr
{
  u_short sa_family;   /* famille d'adresse, 2 bytes */
  char    sa_data[14]; /* 14 bytes d'adresse max */
};
```

Domaine UNIX (AF_UNIX)

⇨ Dans la même machine

```
struct sockaddr_un
{
    short sun_family;
    char sun_path[108]
};
```

Domaine Internet (AF_INET)

⇨ Dans des machines distantes

```
struct in_addr{
  u_long s_addr; };

struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr s_addr;
    char sin_zero[8]};
```

Types de sockets

- SOCK_DGRAM : transport de données en mode non connecté. Dans le domaine AF_INET, le protocole UDP est utilisé.
- SOCK_STREAM : transport de données en mode connecté. Dans le domaine AF_INET, utilisation de TCP. Frontières de messages préservées et possibilité d'envoi de messages urgents.
- SOCK_RAW : permet l'accès au service de niveau 3 (émission/réception de paquets IP)
- SOCK_SEQPACKET : comme le type SOCK_STREAM mais ne permettant pas l'envoi de messages urgents

Structures de données

- Domaine internet : AF_INET

```
#include<sys/types.h>
#include <sys/socket.h>

    struct sockaddr{
        unsigned short sa_family;      //famille de protocole pour cette adresse
        char sa_data[14];               // 14 octets d'adresse
    }
    struct sockaddr_in{                // _in pour Internet
        short sin_family;               //famille de protocole pour cette adresse
        u_short sin_port;               //numéro de port (0=port non utilisé)
        struct in_addr sin_addr;       //adresseIP
        char sin_zero[8];              //non utilisé
    }
    struct in_addr{
       u_long s_addr;          //soit 4 octets : bien pour une adresse IP !
};
```

Structures de données (2)

- Domaine internet : AF_INET

```
struct hostent {
    char *h_name;           /* Nom officiel de l'hôte. */
    char **h_aliases;       /* Liste d'alias. */
     int h_addrtype;        /* Type d'adresse de l'hôte. */
     int h_length;          /* Longueur de l'adresse. */
    char **h_addr_list;     /* Liste d'adresses. */
     }
    #define h_addr h_addr_list[0]    /* pour compatibilité. */
 };
```

Des primitives utiles

- Conversion Network Byte Order(*Big Endian*) – Host Byte Order
  - ***htons():*'Host to Network Short'**
  - ***htonl():*'Host to Network Long'**
  - ***ntohs():*'Network to Host to Short'**
  - ***ntohl():*'Network to Host to Long'**
- ATTENTION: toujours mettre les octets dans l'ordre 'Network Order' avant de les envoyer sur le réseau
- ***in_addr inet_addr(char *)***
  - Convertit une adresse 'ASCII' en entier long signé (en Network Order)

    Exemple : socket_ad.sin_addr.s_addr= inet_addr("172.16.94.100")
- ***char * inet_ntoa(in_addr)***
  - Convertit un entier long signé en une adresse 'ASCII'

```
Exemple :   char *ad_ascii;
            ad_ascii=inet_ntoa(socket_ad.sin_addr),
            printf("adresse: %s\n",ad_ascii);
```

Des primitives utiles (2)

- ***struct hostent gethostbyname(char *name)***
  - Pour traduire un nom de domaine en adresse IP

```
struct hostent*h;
h=gethostbyname("isitcom.rnu.tn");
printf("adresse IP: %s\n",  inet_ntoa(*((struct in_addr*)h->h_addr)));
```

- ***getservbyname()***
  - Pour traduire en n° de port le nom d'un service
- ***getsockname(int desc, struct sock_addr* p_adr, int* p_longueur)***
  - Pour récupérer l'adresse d'une socket(après bind)

Création et fermeture de socket

- ***int socket(int af, int type, int protocole)***
  - Création de la structure de donnée (appelée socket) permettant la communication,
  - af= famille de protocole (TCP/IP, ou d'autres...)
    - AF_INET :domaine Internet (domaine que nous utiliserons)
    - AF_UNIX :domaine UNIX (pour donner un autre exemple)
  - type = SOCK_STREAM, SOCK_DGRAM, RAW
  - Protocole : 0 pour protocole par défaut (voir <netinet/in.h>)
  - socket() retourne
    - un descripteur de socket
    - -1 si erreur
- ***close(int socket)***
  - Ferme la connexion et supprime la structure de données associée à la socket
- ***Shutdown(int socket, int how)***
  - how: 0/1/2 pour réception interdite/émission interdite/réception&émission interdite

Association de socket (*bind)*

- ***int bind(int socket, struct sockaddr* adresse-locale, int longueur-adresse)***
  - Associe un numéro de port et une adresse locale à une socket, retourne –1 si erreur.
  - socket= descripteur de socket
  - adresse-locale = structure qui contient l'adresse (adresse IP + n° de port)
  - adresse-locale: struct sockaddr*
    - sockaddr_un si AF_UNIX
    - sockaddr_in si AF_INET(adresse IP)…
  - longueur adresse : sizeof(struct sock_addr)
  - Si sin_port=0: choix d'un numéro de port non utilisé
  - Si sin_addr.s_addr= INADDR_ANY: socket affectée à toutes les interfaces locales

Ecoute/ Acceptation des connexions entrantes

- ***int listen (int desc, int nb)***
  - Permet de mettre une socket serveur en attente de clients
    - desc : descripteur du socket d'écoute
    - nb : nombre max de connexions à accepter
  - Ne s'utilise qu'en mode connecté

```
if (listen(socket, 10) == SOCKET_ERROR)
 { // traitement de l'erreur }
```

- ***int accept (int desc, struct sockaddr * p_adr, p_int *lgadr)***
  - desc : descripteur du socket
  - p_adr : adresse du socket connecté du client
  - p_lgadr : longueur de l'adresse connectée

```
sockaddr_in  appelant;  int  len;
If (accept(socket_locale,(struct sockaddr*)appelant, *len) !=INVALID_SOCKET)
{…}
```

Connexion des clients

- ***int connect (int desc, struct sockaddr * p_adr, int lgadr)***
  - Permet d'établir un connexion avec un serveur
    - desc : descripteur du socket locale
    - p_adr : adresse du socket distant
    - lgadr : longueur de l'adresse distante
  - Les deux sockets doivent avoir le même type et famille

```
if (connect(sc,(struct sockaddr*)to ,sizeof(to)) == -1) {
   // Traitement de l'erreur;
 }
```

Envoi de données en mode non connecté

- ***int sendto(int socket, char * buffer, int len, int flags, sockaddr * to, int tolen)***
  - socket : descripteur de la socket locale
  - buffer : un tampon contenant les octets à envoyer
  - len : nombre d'octets à envoyer
  - flags : type d'envoi à adopter (normalement à zéro)
  - to : l'adresse d'une structure qui contient l'adresse du destinataire
  - tolen : la taille de la structure de l'adresse du destinataire

- La fonction sendto() renvoie le nombre d'octets effectivement envoyés.

```
retour = sendto(socket, buffer, sizeof(buffer),0 ,(sockaddr*)to,  sizeof(to));
if (retour == SOCKET_ERROR) {
    // traitement de l'erreur
}
```

Réception de données en mode non connecté

- ***int recvfrom(int socket,char *buf, int len, int flags, sockaddr * from, int * frlen)***
  - socket : descripteur de la socket locale
  - buf : un tampon qui recevra les octets en provenance de l'émetteur
  - len : nombre d'octets à lire
  - flags : type de lecture à adopter (normalement à zéro)
  - from : adresse d'une structure qui contiendra l'adresse de l'émetteur
  - frlen : taille de la structure de l'adresse de l'émetteur

- La fonction recvfrom() renvoie le nombre d'octets lus.
- Cette fonction bloque le processus jusqu'à ce qu'elle reçoive des données

```
retour = recvfrom(socket, buffer, sizeof(buffer),0, (sockaddr*)from, &len);
if (retour == SOCKET_ERROR) {
    // traitement de l'erreur
}
```

Envoi de données en mode connecté

- ***int write(int socket,char * buffer,int len)***
- ***int send(int socket,char * buffer,int len, int flags)***
  - socket : descripteur de la socket locale
  - buffer : tampon contenant les octets à envoyer au destinataire
  - len : nombre d'octets à envoyer
  - flags : type d'envoi à adopter (normalement à zéro)

- Les fonctions write() et send() renvoient le nombre d'octets effectivement envoyés

```
retour = write(socket ,buffer,sizeof(buffer));
//retour = write(socket ,buffer,sizeof(buffer), 0);
if (retour == SOCKET_ERROR) { // traitement de l'erreur }
```

Réception de données en mode connecté

- ***int read(int socket,char * buffer,int len)***
- ***int recv(int socket,char * buffer,int len,int flags)***
  - socket : descripteur de la socket locale
  - buffer : un tampon qui recevra les octets en provenance de l'émetteur
  - len : nombre d'octets à lire
  - flags : type de lecture à adopter (normalement à zéro)

- Les fonctions read() et recv() renvoient le nombre d'octets effectivement lus.
- Ces fonctions bloquent le processus jusqu'à ce qu'elle reçoivent des données

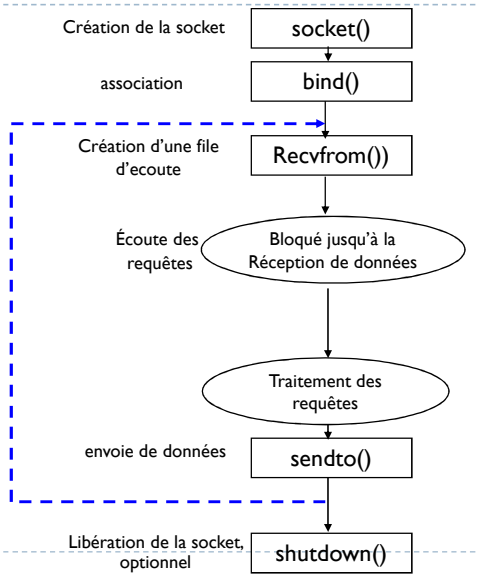
```
retour = read(socket,buffer,sizeof(buffer));
retour = recv(socket,buffer,sizeof(buffer), 0);
if (retour == -1) { // traitement de l'erreur }
```

Types de serveurs

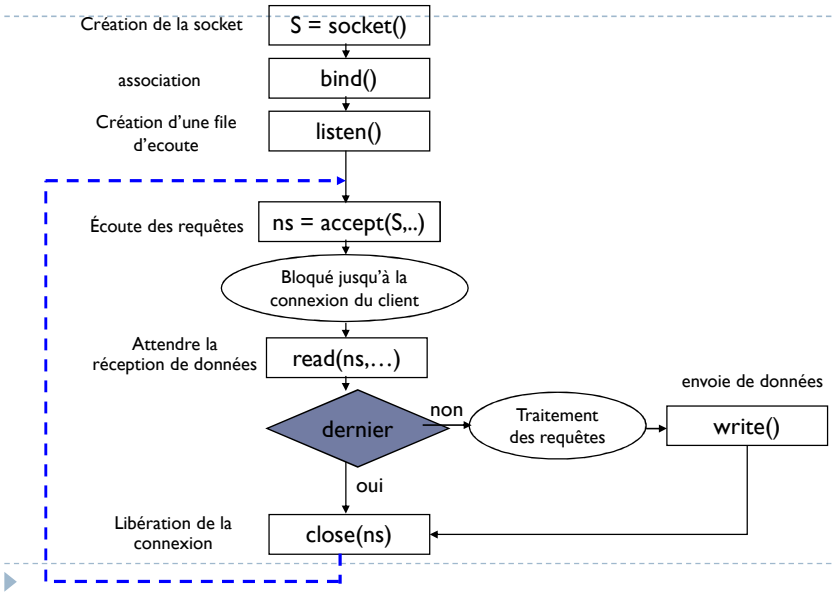
Critère	Type	Description
Type de connexion	Orienté connexion	Établissement d'une connexion puis transmission puis déconnexion entre le serveur et le client.
	Non orienté connexion	Transmission sans connexion au début et sans déconnexion à la fin.
Concurrence	Iteratif	Il y a un seul serveur qui traite un seul client à chaque instant.
	Concurrent	Il y a plusieurs instances de serveurs qui traitent simultanément plusieurs clients.

- Un serveur concurrent orienté connexion s'emploie dans des applications types dans lesquelles un client établit une connexion avec le serveur qui dure beaucoup de temps (telnet, FTP, etc). Dans ce cas, il y a un procesus serveur qui entend à chaque client.
- Un serveur itératif non orienté connexion, s'utilise quand le service sollicité reste un peu de temps (echo, time…etc).

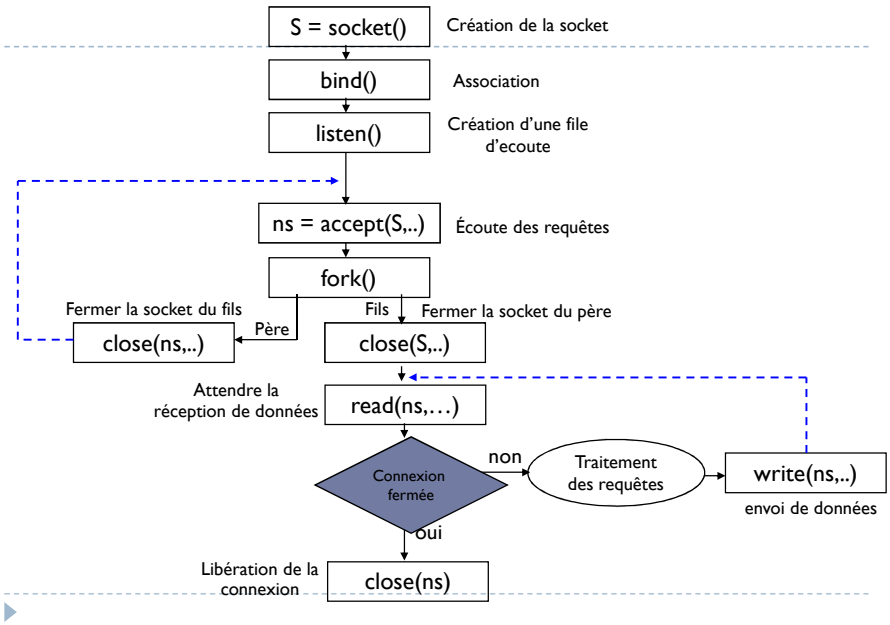
Serveur itératif en mode non connecté



Serveur itératif en mode connecté



Serveur concurrent en mode connecté



- https://slideplayer.fr/slide/472865/
- https://www.journaldunet.com/solutions/cloud-computing/1206814-comment-mettre-en-oeuvre-une-blockchain/