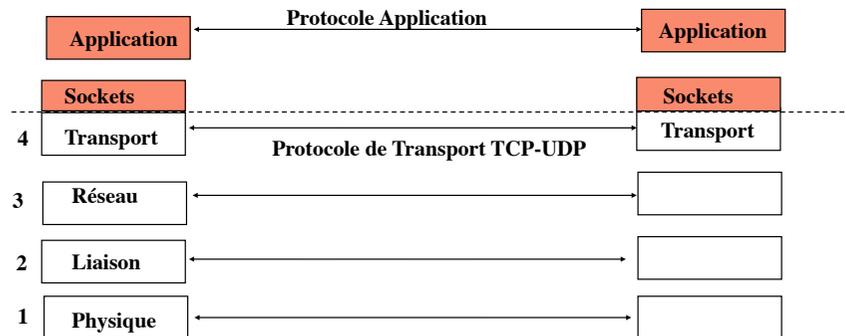


## Programmation en C d'une application client-serveur à l'aide des Sockets



## Principes

- **Socket** : étymologie: prise (de courant,...)
- **Interface des "sockets"**: bibliothèques de primitives (fonctions/procédures) d'accès aux deux protocoles transport d'Internet : TCP et UDP
- Développée par l'université de Berkeley (on parle de Berkeley Socket Interface)
- Disponible dans différents langages (C, Java ...)
- **Cette interface permet la programmation d'applications client/serveur:**
  - Deux programmes différents
    - » Le **serveur** se met en attente de demandes (passif)
    - » Le **client** initie le dialogue par une demande (actif)

## Références

- **Documentations** dans votre poly de Réseau ou plus détaillée sur le Modde de l'UFR IM2AG
- **Livres**
  - J. M. Rifflet, J.-B. Yunès. *Unix - Programmation et Communication*, Dunod (2003), chap. 19
  - R. Stevens. *Unix Network Programming*. Prentice-Hall.
- **Web**
  - <https://fr.wikipedia.org/wiki/Socket>
  - <http://sdz.tdct.org/sdz/les-sockets.html>
- **man** de Unix

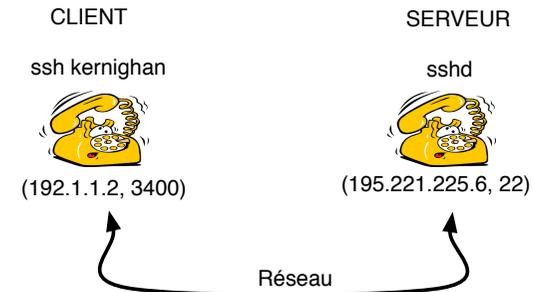
## Analogie avec le téléphone

- Une socket est un point d'accès au réseau que l'on peut comparer à un téléphone
- C'est l'extrémité d'un canal de communication permettant l'échange de donnée entre deux entités (les utilisateurs du téléphone)
- Les applications qui utilisent ces sockets sont les utilisateurs des téléphones
- Un utilisateur : un processus s'exécutant sur un ordinateur
- Un utilisateur demandeur : c'est le client, c'est lui qui compose le numéro
- Un utilisateur en attente de coup de fil: c'est le serveur, il décroche
- Une fois la communication établie, elle est bi-directionnelle et symétrique

## Exemple d'utilisation d'une application client/serveur

- Application ssh
- /etc/services:
  - ssh 22/udp # SSH Remote Login Protocol
  - ssh 22/tcp # SSH Remote Login Protocol
- ssh kernighan.imag.fr
  - On lance le client en lui donnant le nom DNS ou l'adresse Internet du serveur à connecter
  - L'application consulte l'annuaire DNS : kernighan.imag.fr ->195.221.225.6
  - Elle demande une connexion avec la socket du serveur (195.221.225.6, 22)
  - La socket côté client est (192.1.1.2, 3400) : numéro de port libre à ce moment sur la machine où tourne le client (192.1.1.2)
- Il faut qu'un serveur ssh soit lancé sur la machine destination
  - Processus (ou démon) sshd (ps -aux | grep sshd)

## Exemple d'utilisation d'une application client/serveur



- Une fois la communication établie, le flux de donnée peut être bi-directionnel
- La communication est identifiée par le quadruplet:
  - (adresse IP source, port source, adresse IP destination port destination)

## Numéros et annuaires

- Identification d'un point d'accès à l'aide du couple:
  - Une adresse Internet: identifie la machine dans le réseau
  - Un numéro de port: identifie l'application sur la machine
- Annuaires disponibles:
  - Local Nom/adresses internet: fichier système sur la machine (/etc/hosts)
  - Global Nom/adresses internet: le DNS (Domain Name System)
    - Distribué
    - Client et serveur DNS
  - Local Application/numéro de port : fichier système sur la machine (/etc/services)
    - Numéro de port attribués aux applications "standards" (réservés < 1024)
    - Ceux sont les numéros de port des sockets des serveurs, on parle aussi de service

## Accès à l'annuaire

- La recherche dans l'annuaire local ou le DNS peut être défini dans un fichier système: /etc/host.conf sous free-BSD
- Récupération de l'adresse à partir du nom (*obsolète*)
  - *struct hostent \*gethostbyname (char \*nom)*
  - retourne un pointeur sur une structure contenant les adresses Internet correspondant au nom
- Récupération du numéro de port à partir du nom du service (donné dans /etc/services) (*obsolète*)
  - *struct servent \*getservbyname(char \*service, char \*protocole)*
  - retourne un pointeur sur une structure contenant le numéro de port correspondant au service et au protocole associé

## Accès à l'annuaire

- La fonction `getaddrinfo()` combine les possibilités offertes par les fonctions `gethostbyname` et `getservbyname` en une unique interface
- **Récupération de l'adresse à partir du nom**
  - `int getaddrinfo(const char *nom, const char *service, const struct addrinfo *hints, struct addrinfo **res);`
  - rempli une ou plusieurs structures (*res*) contenant les adresses Internet correspondant au *nom* et au *service* (numéro de port)
  - la structure hints peut indiquer la valeur de certains champs:
    - par exemple:
      - `hints.ai_family = AF_UNSPEC; /* Allow IPv4 or IPv6 */`
      - `hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */`

## La structure addrinfo

```
struct addrinfo {
    int         ai_flags;
    int         ai_family;
    int         ai_socktype;
    int         ai_protocol;
    size_t      ai_addrlen;
    struct sockaddr *ai_addr;
    char        *ai_canonname;
    struct addrinfo *ai_next;
};
```

La structure `sockaddr` peut ensuite être utilisée pour les appels aux primitives `bind`, `connect` ... (voir plus loin)

## Accès à son adresse et numéro de port

- **Récupération de sa propre adresse:**
  - Permet de définir la socket locale
  - `int gethostname(char *nom, int longueur_nom)`
    - renseigne le nom de la machine sur laquelle s'exécute la procédure
  - On utilise ensuite `gethostbyname` pour avoir l'adresse associée
- **Récupération d'un numéro de port alloué dynamiquement**
  - `int getsockname(int socket, struct sockaddr_in *p_ad_s, int *len)`
  - *\*len* doit contenir la longueur de la structure `sockaddr_in`, attention c'est un paramètre donnée/résultat

## Installer une nouvelle prise de téléphone

- La fonction `int socket(int domaine, int mode, int protocole)`
- Elle retourne un identificateur de socket (entier qui est un descripteur de fichier), -1 en cas d'erreur
  - Cet identificateur est locale à la machine et n'est pas connu par le destinataire
- **domaine:**
  - `AF_INET` : prise Réseau Internet IPV4
  - `AF_INET6` : prise Réseau Internet IPV6
  - `PF_INET` : prise pour utilisation locale (interne à la machine entre processus)
- **mode:**
  - `SOCK_STREAM` (pour TCP), `SOCK_DGRAM` (pour UDP), `SOCK_RAW` (pour IP)
- **protocole:**
  - associé au mode (`IPPROTO_UDP`, `IPPROTO_TCP`, `IPPROTO_RAW`, `IPPROTO_ICMP`)

## Associer un numéro à la prise de téléphone

- La fonction `int bind( int sock, sockaddr_in *p_adr_s, int lg_struct)`
  - `sock`: identificateur de la socket
  - `p_adr_s`: pointeur vers une structure contenant les numéros à associer à la socket (@IP et port)
  - `lg_struct`: la longueur de la structure

- Structure `sockaddr_in`
  - A remplir avant l'appel de `bind`

```
- struct sockaddr_in
{
    short sin_family; /* famille d'adresse */
    ushort sin_port; /* numéro de port */
    ulong sin_addr; /* adresse de niveau 3 : IP*/
    char sin_zero [8]; /* inutilisé (mis à zéro) */
}
```

AF_INET
3400
192.1.1.2

## Numéros spéciaux

- Il est possible de laisser au système le choix d'un numéro de port libre au moment de l'appel à la fonction `bind`
  - C'est intéressant dans le cas d'un client
  - Il suffit de mettre dans la structure `sockaddr_in` le champ `sin_port` à 0
- Il est possible d'associer à une socket l'ensemble des adresses IP de la machine dans le cas où elle est connectée à plusieurs réseaux
  - Cela permet dans le cas d'un serveur d'être accessible via ces différents réseaux
  - Il suffit de remplir dans la structure `sockaddr_in` le champ `sin_addr` par la constante `NULL`

## Allocation des sockets et affectation des numéros

CLIENT

`socket`  
`bind 192.1.1.2, 0`

Identificateur: 3



(192.1.1.2, 3400)

SERVEUR

`socket`  
`bind NULL, 22`

Identificateur: 5



(\* , 22)

## Appel d'un numéro et établissement de la communication par le client

- Il faut préciser les numéros du destinataire (serveur)
- Il faut remplir une structure `sockaddr_in` avec les numéros du destinataire
- Puis appeler la fonction d'établissement de communication qui dépend du protocole (UDP ou TCP)

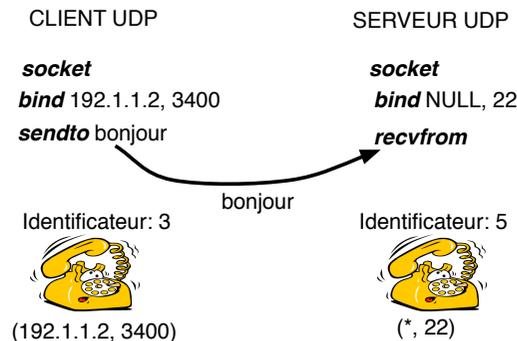
## L'établissement de la communication par le client dans le cas d'UDP

- Pas d'établissement de connexion préalable, on envoie le premier paquet de donnée
- **Fonction** *int sendto (int sock, char \*data, int lg\_data, struct sockaddr\_in \*p\_ad\_s, int flags, int lg\_struct)*
  - **Data**: les données, tableau (ou chaîne) de caractères à envoyer
  - **Lg\_data** : on précise leur longueur
  - **p\_ad\_s** : structure contenant les numéros du destinataires
  - **Flags** : non utilisé, à mettre à 0
  - Retourne -1 si il y a une erreur, sinon le nombre d'octets **effectivement envoyés**

## Attente d'appel sur un téléphone Cas du serveur avec UDP

- Comme pour le client il faut mettre en place la prise (socket) et lui affecter des numéros (bind)
- Ensuite il faut se tenir prêt à recevoir des coups de fil des clients
- Dans le cas de UDP, cela se manifeste par l'envoi d'un premier paquet de donnée
- **La fonction** *int recvfrom (int sock, char \*data, int lg\_data, struct sockaddr\_in \*p\_ad\_s, int flags, int lg\_struct)*
  - **Data**: Le tableau (ou chaîne) de caractères reçu (rempli par la fonction)
  - **Lg\_data** : on précise leur longueur
  - **p\_ad\_s**: structure qui est remplie par recvfrom et qui contient les numéros du client qui vient d'envoyer ces données
  - Retourne le nombre d'octet effectivement reçu, -1 s'il y a une erreur

## Un appel téléphonique avec UDP



- Le récepteur possède un buffer d'une taille fixe
- Une fois le premier paquet émis le serveur peut aussi envoyer des données au client

## L'établissement de la communication par le client dans le cas de TCP

- Il y a l'établissement d'une connexion préalable, les données seront envoyées ensuite
- **Fonction** *int connect (int sock, struct sockaddr\_in \*p\_ad\_s, int lg\_struct)*
  - **p\_ad\_s** : structure contenant les numéros du destinataires
- Si l'établissement de la connexion par TCP est réussie, la fonction retourne 0 sinon -1

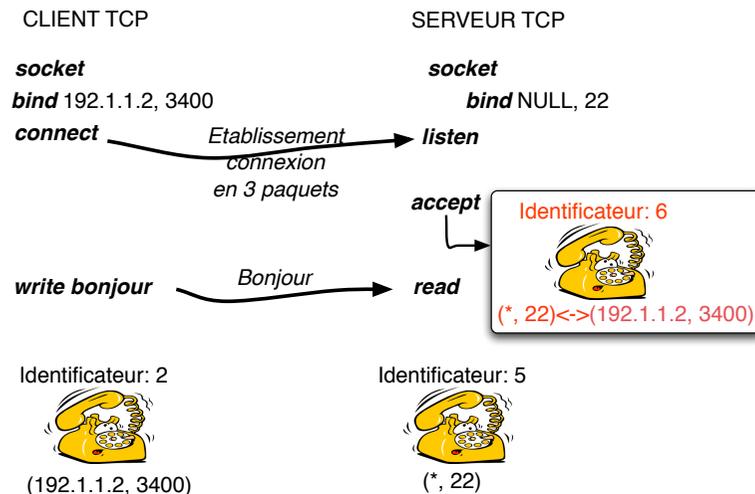
## Attente d'appel sur un téléphone Cas du serveur avec TCP

- Comme pour le client il faut mettre en place la prise (socket) et lui affecter des numéros (bind)
- Ensuite il faut se tenir prêt à recevoir des coups de fil des clients
- **La fonction `int listen ( int sock, int nb_requete)`**
  - **sock** : identificateur de la socket dont on renseigner précédemment les numéros
  - **nb\_requete** : nombre maximale de requête pouvant être mémorisé en attendant un traitement par le serveur

## Acceptation d'un appel sur un téléphone Cas du serveur avec TCP

- Le serveur peut accepter une demande de communication sur une socket précédemment mise en attente (par `listen`)
- **La fonction `int accept ( int sock, struct sockaddr_in *p_ad_s, int lg_struct)`**
  - **sock** : identificateur de la socket en attente
  - **p\_ad\_s**: structure qui est remplie par `recvfrom` et qui contient les numéros du client qui vient d'envoyer ces données
  - cette fonction est bloquante si il n'y a pas eu de demande de connexion de client
- **ATTENTION:** elle retourne un nouvel identificateur de socket mais portant les même numéros (port et adresse).
  - C'est sur cette nouvelle socket que peuvent se faire les échanges de données
  - **La socket initiale (passive) est toujours en attente de demande de connexion**
  - Permet de faire des serveurs multi-clients

## Un appel téléphonique avec TCP



## On parle dans le téléphone Cas de TCP

- Une fois la connexion établie le client peut envoyer et recevoir des données, le serveur aussi (après l'`accept`)
- **La fonction `int write ( int sock, char *data, int lg_data)`**
  - **Data**: Le tableau (ou chaîne) de caractères reçu (rempli par la fonction)
  - **Lg\_data** : Taille du tableau
  - Retourne le nombre d'octets effectivement envoyés, -1 s'il y a une erreur
- **La fonction `int read ( int sock, char *data, int lg_data)`**
  - **Data**: Le tableau (ou chaîne) de caractères à envoyer
  - **Lg\_data** : Taille du tableau
  - Retourne le nombre d'octets effectivement reçus, -1 s'il y a une erreur

## Le problème de la langue

- Les données ne sont pas représentées de la même façon suivant les processeurs (little-big endian)
- Il faut passer par un traducteur avant de les envoyer sur le réseau
- **La fonction *short int htons (short int x)***
  - home to network
  - retourne l'entier court (2 octets) à la norme réseau de x (passé en norme machine)
- **La fonction *long int htonl (long int x)***
  - retourne l'entier long (4 octets) à la norme réseau de x
- **La fonction *short int ntohs (short int x)***
  - network to home
  - retourne l'entier court (2 octets) à la norme machine de x (passé en norme réseau)
- **La fonction *long int ntohl (long int x)***
  - retourne l'entier long (4 octets) à la norme réseau de x

## Fin de la communication Cas de UDP et TCP

- **Il faut maintenant raccrocher le téléphone**
- **La fonction *int close (int sock)***
  - Retourne -1 si il y a une erreur
  - Fermeture complète, on ne peut plus envoyer ou recevoir de données
  - La connexion est complètement fermée et libérée une fois que le client et le serveur on fait close
  - la fermeture est symétrique: le client ou le serveur peut commencer la fermeture
- **La fonction *int shutdown (int sock, int sens)***
  - On peut préciser le sens de fermeture, la connexion n'est alors pas complètement fermée
  - **Sens:**
    - 0 fermeture en entrée
    - 1 fermeture en sortie
    - 2 fermeture dans les deux sens: équivalent à un close
  - Retourne -1 si il y a une erreur, 0 sinon

## Les options

- On peut consulter/modifier des options sur des sockets allouées
- **Exemples**
  - Taille du buffer de réception, d'émission
  - Définition de socket multicast
  - Réception/émission de paquets broadcast
  - Priorité
  - ...
- **Fonction *int getsockopt***
  - Consultation des options en cours
- **Fonction *int setsockopt***
  - Modification des options

## En Travaux pratiques

- Librairie "encapsulée" des primitives des sockets (fichier **fon.c**, **fon.h**)
- **Exemple : *bind*** appelé dans ***h\_bind***
- Simplification des appels des primitives (moins de paramètre)
- Evite de manipuler la structure `sockaddr_in` grâce à la procedure ***adr\_socket (char \*service, char \*adresse, int TypeSock, struct sockaddr\_in \*\*pp\_ad\_s)***
  - **service** : numéro de port ou nom associé dans /etc/services (Ex: "3400")
  - **adresse**: adresse en décimal pointé ou nom associé (/etc/hosts ou DNS) (ex: "192.1.1.2" ou " www.ufrima.fr ")
  - **TypeSock**: **SOCK\_DGRAM** (UDP) ou **SOCK\_STREAM** (TCP)
  - **pp\_ad\_s**: pointeur sur pointeur sur la structure qui est allouée par la procédure
  - **Exemple**: ***adr\_socket("3400", "192.1.1.2", SOCK\_STREAM, &psocclient);***

## En Travaux pratiques Compilation et debugage

- **client.c et serveur.c “prêt à l’emploi”**
  - Récupération des arguments passés au moment du lancement
    - client 192.0.1.1 2200 udp
    - serveur 2200 udp
- **makefile** pour compiler fon.c, fon.h avec client.c et serveur.c
- **Commandes: make, make clean**
- Mode debug possible (voir dans le *makefile* : -DDEBUG) pour avoir des traces d’exécution à l’écran des primitives des sockets
- Pour tester votre client indépendamment du serveur, vous pouvez utiliser **socklab** pour dialoguer avec votre programme

## Serveur à traitement itératif des clients

- **Cas de TCP :**
  - les demandes de connexion sont mémorisés par TCP en attente de traitement
  - La socket passive est toujours à l’écoute
  - Les connexions sont traitées les unes après les autres à travers la socket générée par la fonction accept

## Serveur à traitement itératif des clients avec TCP

CLIENTS



(192.1.1.2, 3400)



(56.1.1.10, 5000)

**socket**  
**bind**  
**connect**  
**read/write**  
**close**

SERVEUR ITERATIF  
un client à la fois

**socket**  
**bind** NULL, 22  
**listen**  
Id: 5



(\* , 22)

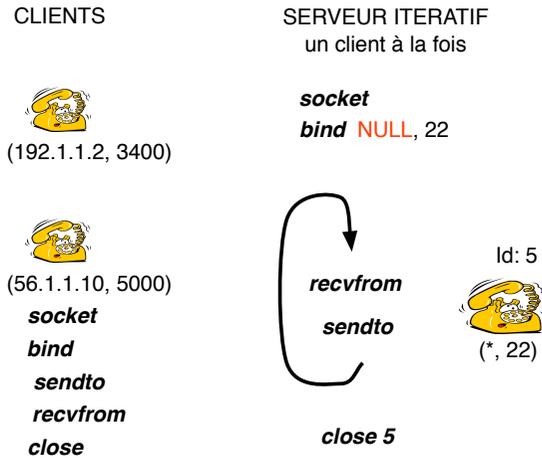
**accept** Id: 6  
**read/write** (\* , 22)<->(192.1.1.2, 3400)  
**close** 6

**close** 5

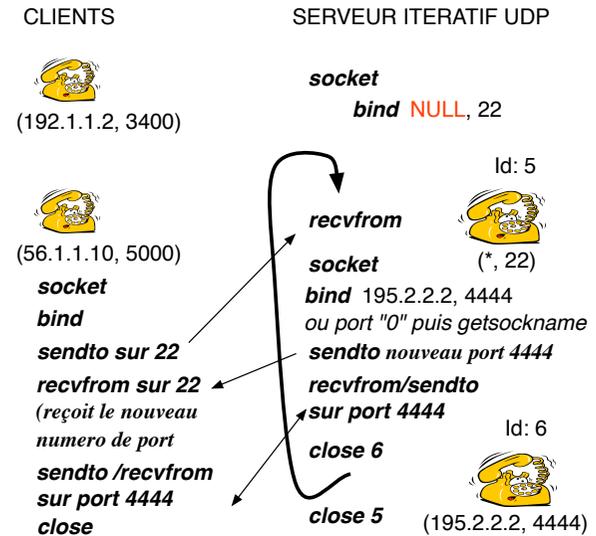
## Serveur à traitement itératif des clients

- **Cas de UDP:**
  - les paquets reçus sont mémorisés dans un buffer en attente de réception
  - Si le dialogue se résume à l’échange de deux paquets (question/ réponse), on traite successivement les demandes
  - Si le dialogue est plus compliqué (dialogue à état)
  - Il va y avoir mélange des demandes de communication et des échanges dans des communications “déjà établies”
  - Il faut alors créer une nouvelle socket à la main (comme le fait l’accept) et continuer le dialogue sur cette nouvelle socket
  - Cette socket est forcément sur un port différent
  - Ce nouveau port doit donc être communiqué au client au moment de sa demande de communication

## Serveur à traitement itératif des clients avec UDP - Question/réponse

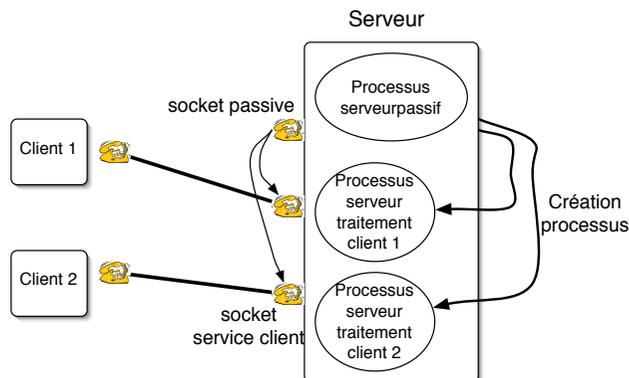


## Serveur à traitement itératif des clients avec UDP - Dialogue/serveur à état



## Serveur à traitement parallèle des clients

- Il faut générer des processus qui vont s'exécuter en parallèle
- Le processus à l'écoute des demandes génère un processus par client



## La création de processus

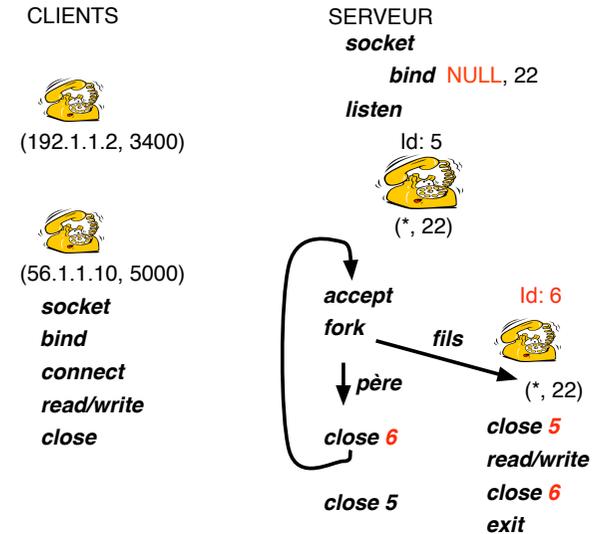
- Fonction **int fork()**
  - Crée un nouveau processus (appelé  *fils* ) à l'identique du processus  *père*
  - Le processus père est dupliqué en mémoire (instruction machines et variables)
  - Il n'y a pas de partage de variable entre les processus
  - Pour les sockets il faut que les deux processus  *père*  et  *fils*  ferme la socket pour qu'elle soit complètement libérée
  - La seule différence entre les deux processus est la valeur retournée par la fonction **fork**
    - Pour le  *père*  elle retourne la valeur du PID (Processus Identifier) du  *fils*
    - Pour le  *fils*  elle retourne 0

## Exemple d'utilisation du fork

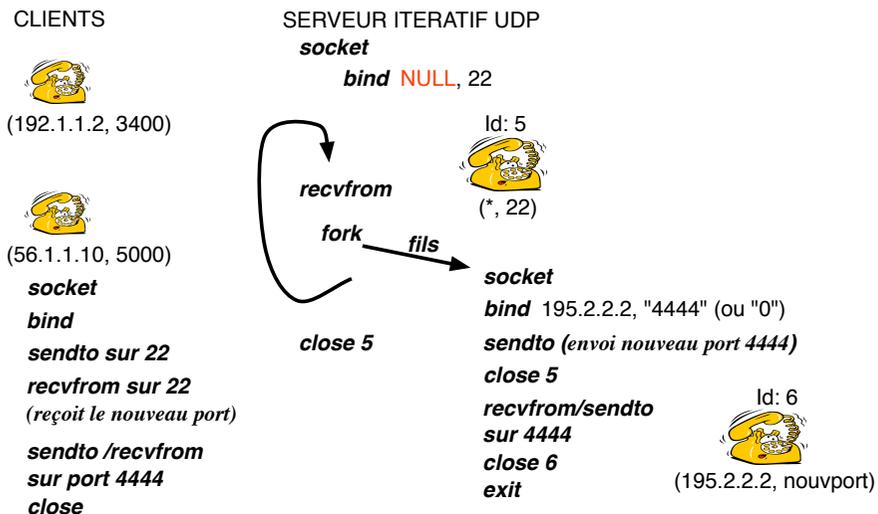
```
int pid;
pid=fork();
if (pid==0) printf("fils\n")
else printf ("père\n");
printf("père et fils\n");
```

- A l'exécution à l'écran:
  - fils
  - père
  - père et fils
  - père et fils
- On ne connaît l'ordre dans lequel ces printf seront exécutés

## Serveur à traitement parallèle des clients avec TCP



## Serveur à traitement parallèle des clients avec UDP



## Attente d'évènement sur plusieurs sockets La fonction *select*

- Autre façon de servir les clients en parallèle
- La fonction *select* permet de se mettre en attente d'évènements sur plusieurs sockets à la fois
- Des fonctions associées et types prédéfinis dans la librairie `<sys/types.h>`

## La fonction *select*

- ***int select(int maxfdpl, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout)***
  - *fd\_set* : type permettant de définir un ensemble de sockets
  - Bloquante tant qu'une des sockets au moins de *readfds* n'est pas en attente de lecture
  - Supprime de l'ensemble de socket *readfds* les sockets qui ne sont pas en attente de lecture
- **Exemple:**

```
fdset set; int idsock1, idsock2, maxsock;

FD_ZERO(&set);          /* initialise set à vide */
FD_SET(idsock1, &set);  /* ajoute idsock1 a set */
FD_SET(idsock2, &set);  /* ajoute idsock2 a set */
maxsock=getdtablesize();
select (maxsock, &set, 0, 0, 0);
if (FD_ISSET(idsock1, &set) ... /*la socket idsock1 a reçu des données*/
```

## Utilisation de la fonction *select*

- Le clavier et l'écran sont associés aux descripteurs de fichier 0 et 1
- On peut donc attendre un événement sur un ensemble de socket et le clavier
- Indispensable par exemple pour faire un talk asynchrone
- En effet les primitives *read* et *getch* (récupération d'un caractère au clavier) sont bloquantes:
  - *car= getch(); write(idsock, &car, 1); /\*bloquer sur le getch jusqu'à que l'on tape un caractère au clavier\*/*
  - ...
  - *nblu= read(idsock, &car, 1); putchar (car); /\*bloquer sur le getch jusqu'à que l'on tape un caractère au clavier\*/*

## Exemple d'utilisation de *select* pour réaliser une application « talk »

```
FD_ZERO(&set); /* initialise set à vide */
FD_SET(0, &set); /*ajoute de la socket « clavier » a set */
FD_SET(idsock, &set); /*ajoute idsock a set */
select (maxsock, &set, 0, 0, 0)
if (FD_ISSET(0, &set) /* un caractère a été tapé au clavier */
    {car1=getch();
    ...
    }
if (FD_ISSET(idsock, &set) /* un caractère a été reçu sur la socket réseau */
    {
    nbcars=read(idsock,&car, lg_message);
    ...
    }
```

## Choix du protocole

- Avantages UDP par rapport à TCP
  - Plus rapide (pas d'établissement de connexion)
  - Moins "coûteux" pour le réseau (petite entête, pas d'autres paquets (*syn, ack, close*))
  - Possibilité de broadcast et multicast
  - Intéressant si communication "question-réponse"
  - Mobilise moins de ressources sur la machine que TCP
- Inconvénients UDP par rapport à TCP
  - Pas de récupération d'erreur ni de contrôle de flux, à gérer par l'application
  - Peu adapté à une communication à "état" (dialogue élaboré) Problème des pertes de paquets dans le dialogue