heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Chapitre IV

La programmation réseau

Auteur: Prof. Dr. Jurgen Ehrensberger

Contenu

- Utiliser les sockets BSD pour la transmission de données
- Développer des applications communicantes sur TCP et UDP
- Gérer des communications concurrentes
- Développer des applications multicast
- Sécuriser la transmission avec SSL/TLS

Sockets BSD

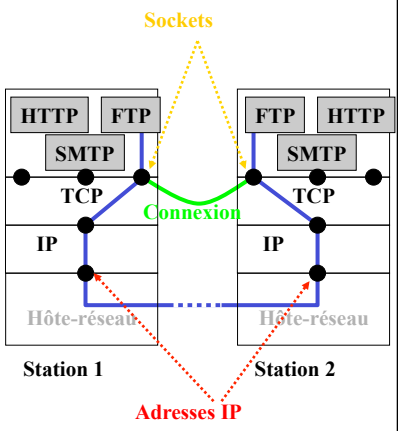
- Utilisés sur les systèmes Unix et Linux pour la communication à travers le réseau
  - Windows utilise les Winsocks qui sont relativement similaires
- Typiquement utilisés avec TCP et UDP
  - *Raw sockets*: transmission avec IP
  - *Device sockets*: accès direct à la couche liaison
- Définissent une API (interface de programmation d'applications)
  - Fonctions pour ouvrir une connexion, envoyer des données, lire les données reçues, …

Socket

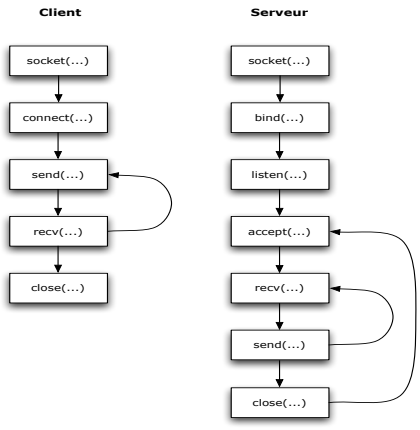
- Un socket est une extrémité d'une communication
- Un socket a un comportement essentiellement similaire à un fichier
  - Ouverture du socket
- L'OS renvoie un descripteur de socket qui est un petit entier qui permet d'identifier le socket
  - read/write sur le socket
- Les appels sont bloquants
  - close du socket

Sockets connectés et non connectés

- TCP: deux sockets doivent être connectés
  - Opérations read/write suivantes n'ont plus besoin d'indiquer le destinataire
- UDP: un socket peut être connecté ou non
  - Connecté: plus besoin d'indiquer le destinataire lors de read/write
  - Non connecté
    - Doit indiquer le destinataire lors d'un envoi
    - Doit examiner la source lors d'une réception

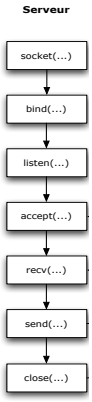


Utilisation de l'API sockets
(mode connecté)



Création d'un socket

sockdesc = **socket**(*family*, *type*, *protocol*)



- Effet:
  - Crée un nouveau socket
- Arguments
  - **Family**: indique la famille de protocoles à utiliser
  - **Type**: type de communication souhaité (fiable ou non)
  - **Protocol**: protocole exact à utiliser
    - Typiquement mis à 0
- Résultat
  - Descripteur de socket: entier sur 32 bits
  - Utilisé dans les appels suivants

Familles d'adresses

Famille	Explication
PF_INET	**Communication à l'aide des protocoles TCP/IP.**
PF_INET6	**Communication sur IPv6**
PF_LOCAL (ou PF_UNIX)	Pour la communication interprocessus sur la même machine à travers des tubes (`pipe` en anglais).
PF_LINK	Protocoles de la couche liaison.
PF_ISDN	Communication à travers ISDN.
PF_PPP	Protocole PPP.
PF_ROUTE	Accès à la table de routage du noyau.
PF_NETBIOS	Protocoles NETBIOS sur Windows.

Types de protocoles de la famille PF_INET

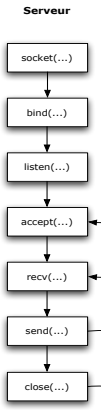
Type de service	Explication
SOCK_STREAM	Service fiable en mode connecté.
SOCK_DGRAM	Service datagramme non fiable, sans connexion.
SOCK_RAW	Accès aux protocoles internes et interfaces (typiquement IP). Accessible uniquement à l'utilisateur privilégie.

- Pour TCP sur IPv4:
  int sd = **socket**(*PF_INET*, *SOCK_STREAM*, *0*)
- Pour UDP sur IPv4:
  int sd= **socket**(*PF_INET*, *SOCK_DGRAM*, *0*)

Spécification de l'adresse locale

result = **bind**(*socket*, *adr_locale*, len_adr)

- Uniquement appelée par un serveur pour définir l'adresse et le port local sur lesquels le serveur écoute
- Effet:
  - Lié le socket à une adresse locale et un port local
- Arguments
  - **socket**: descripteur de socket
  - **adr_locale**: structure générique qui contient l'adresse
  - **len_adr**: longueur de la structure
- Résultat
  - 0 si succès, sinon -1



La structure d'adresses socket

- Structure sockaddr:
  - Structure générique qui permet de définir les adresses pour n'importe quelle famille de protocoles
- Structure sockaddr_in
  - Redéfinit sockaddr pour les adresses IP

```
struct sockaddr_in {
        __uint8_t       sin_len;      // total length
        sa_family_t     sin_family;   // address family
        in_port_t       sin_port;     // TCP/UDP port number
        struct  in_addr sin_addr;     // IP address
        char            sin_zero[8]   // not used (set to zero)
};
```


Utilisation de sockaddr_in

```
struct sockaddr_in {
        __uint8_t       sin_len;      // total length
        sa_family_t     sin_family;   // address family
        in_port_t       sin_port;     // TCP/UDP port number
        struct  in_addr sin_addr;     // IP address
        char            sin_zero[8]   // not used (set to zero)
};
```

- **sin_family**: indique la famille d'adresses
  - AF_INET pour la famille de protocoles PF_INET
- **sin_port**: numéro de port
  - En représentation réseau standard (Big Endian)
- **sin_addr**: adresse IP en format numérique
  - Ou la constante symbolique INADDR_ANY pour couvrir toutes les adresses locales d'une machine

Manipulation d'adresses IP

- Format décimal pointé → format numérique

  result = **inet_pton**(AF_INET, "111.2.3.4", &sockaddr.sin_addr)

  – Résultat:
    - 1 si succès, 0 si adresse non valable, -1 si erreur

- Format numérique → format décimal pointé

  result = **inet_ntop**(AF_INET, sockaddr.sin_addr, &buffer, bufflen)

  – Résultat:
    - Pointeur NULL si erreur, sinon &buffer

Manipulation de numéros de port

- Les numéros de ports doivent être exprimés en représentation réseau standard (Big Endian)
- Conversion host → réseau pour short int

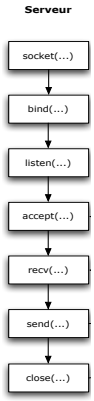
  net_short = **htons**(host_short)
- Conversion réseau → host pour short int

  host_short = **ntohs**(net_short)
- Fonctions similaires pour long int

Mettre le socket en mode passif

result = **listen**(*socket*, *queue_len*)

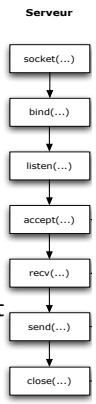
- Uniquement effectué par le serveur pour un socket du type STREAM
- Effet:
  - Crée une file d'attente pour les demandes d'établissement de connexion
- Arguments
  - **socket**: descripteur de socket
  - **queue_len**: longueur de la file d'attente (typiquement 5-20)
- Résultat
  - 0 si succès, sinon -1



Accepter une nouvelle connexion

newsock = **accept**(*socket*, *sockaddr*, *addr_len*)

- Uniquement effectué par le serveur, après listen (socket STREAM)
- Effet:
  - Bloque le serveur jusqu'à l'arrivée d'une demande de connexion
  - Crée un nouveau socket
  - Le socket passif reste ouvert et reçoit des demandes de connexion
- Arguments
  - **socket**: descripteur du socket passif
  - **sockaddr**: pointeur vers la structure qui contiendra l'adresse du client
  - **addr_len**: longueur de la structure sockaddr
- Résultat
  - Descripteur de socket (>=0) si succès, sinon -1

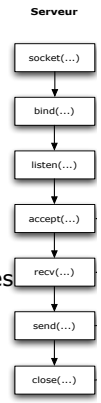


Recevoir et envoyer des données

- Différentes fonctions sont disponibles
- Sockets connectés (TCP ou UDP)
  - **read** et **write** : fonctions Unix habituelle d'E/S
  - **recv** et **send** : permet de spécifier des options supplémentaires
  - readv et writev : variante avec un vecteur de blocs mémoire
- Sockets non connectés (UDP)
  - L'adresse du nœud distant doit être gérée pour chaque appel
  - **recvfrom** et **sendto** : fonctions les plus souvent utilisées
  - recvmsg et sendmsg : utilise une structure qui contient le message et l'adresse paire

Réception de données - socket connecté

nbytes= **read**(*socket*, *buffer*, *maxbytes*)



- Uniquement avec un socket connecté
- Effet:
  - Bloque le programme jusqu'à l'arrivée de données
  - Copie les données reçues dans la zone mémoire *buffer*
- Arguments
  - **socket**: descripteur de socket connecté
  - **buffer**: pointeur vers la zone mémoire où seront copiées les données
  - **maxbytes**: nombre maximum d'octets à copier
- Résultat
  - Si succès, nombre d'octets effectivement lus
  - 0 si la connexion a été fermée (End-of-file)
  - -1 si erreur

Envoi de données - socket connecté

nbytes= **write**(*socket*, *buffer*, *maxbytes*)

- Uniquement avec un socket connecté
- Effet:
  - Bloque le programme jusqu'à ce que l'envoi soit possible (contrôle de flux!)
  - Envoie les données de la zone mémoire *buffer*
- Arguments
  - **socket**: descripteur de socket connecté
  - **buffer**: pointeur vers la zone mémoire avec les données à envoyer
  - **maxbytes**: nombre maximum d'octets à envoyer
- Résultat
  - Si succès, nombre d'octets effectivement envoyés
  - -1 si erreur

Réception de données - socket non connecté

nbytes= **recvfrom**(*socket*, *buffer*, *maxbytes*
flags, addr_src, addr_len)

- Typiquement avec un socket non connecté
- Effet:
  - Bloque le programme jusqu'à l'arrivée de données
  - Copie les données reçues dans la zone mémoire *buffer*
  - Remplit *sockaddr* avec l'adresse de l'émetteur
- Arguments
  - **socket, buffer, maxbytes**: comme pour *read*
  - **flags**: options supplémentaires comme MSG_DONTWAIT ou MSG_PEEK
  - **addr_src**: va être remplie avec l'adresse de l'émetteur (*struct sockaddr*)
  - **addr_len** : longueur de la structure *addr_src*, modifiée par la fonction
- Résultat : comme pour read

Envoi de données - socket non connecté

nbytes= **sendto**(*socket*, *buffer*, *maxbytes*, *flags, addr_dst, addr_len*)

- Typiquement avec un socket non connecté
- Effet:
  - Bloque le programme jusqu'à ce que l'envoi soit possible (contrôle de flux!)
  - Envoie les données de la zone mémoire *buffer*
- Arguments
  - **socket, buffer, maxbytes**: comme pour *write*
  - **flags**: options supplémentaires, comme p.ex MSG_OOB pour données urgentes en TCP
  - **addr_dst** : contient l'adresse du destinataire
  - **addr_len** : longueur de la structure *addr_dst*
- Résultat: comme pour *write*

Fermeture du socket

result= **close**(*socket)*

- Pour sockets connectés ou non connectés
- Effet:
  - Ferme le socket et le détruit
  - En mode STREAM, libère également la connexion
- Arguments
  - **socket**: descripteur de socket
- Résultat
  - 0 si succès,
  - -1 si erreur

Fermeture du socket – mode STREAM

result= **shutdown**(*socket, option)*

- Permet de mieux contrôler la fermeture de connexions TCP
- Utilisé si un côté de la connexion a fini de transmettre
- Effet:
  - Libère la connexion pour l'écriture, lecture ou les deux
  - *Close* doit être appelé après la fermeture par le pair
- Arguments
  - **socket**: descripteur de socket
  - **option**: peut être SHUT_RD, SHUT_WR ou SHUT_RD_WR
- Résultat
  - 0 si succès,
  - -1 si erreur

Connexion d'un socket à une adresse de destination

result= **connect**(*socket, addr_dst, addr_len)*

- Utilisé par un client
- Applicable aux sockets DGRAM ou STREAM
- Socket DGRAM: évite de spécifier le destinataire pour chaque *write()*
- Effet:
  - Lie un socket d'un client à un socket du serveur
  - Socket STREAM: établit une connexion TCP
- Arguments
  - **socket**: descripteur de socket
  - **addr_dst** : contient l'adresse du destinataire (*struct sockaddr*)
  - **addr_len** : longueur de la structure *addr_dst*
- Résultat
  - 0 si succès, -1 si erreur

Architecture d'applications client-serveur

- Algorithme d'un serveur mono-tâche
  - Créer un socket
  - Le lier à un port local
  - Boucle: Accepter une requête et renvoyer la réponse
- Ce comportement n'est acceptable pour pour des services simples, avec peu de clients et des réponses très courtes
- Un serveur réaliste doit être capable de gérer plusieurs clients en parallèle

Serveur itératif sans connexion	Serveur itératif avec connexion
Serveur concurrent sans connexion	Serveur concurrent avec connexion

Serveurs itératifs et concurrents

Serveur itératif

- Traite un seul client à la fois
- Le deuxième client doit attendre la fin du service du premier client

➢ Facile à développer
➢ Simple à comprendre
➢ Performances insuffisantes si les requêtes arrivent en parallèle et que le traitement des requêtes est long

Serveur concurrent

- Traite plusieurs communications en parallèle
- Utilise plusieurs threads, processus ou d'autres mécanismes

➢ Evite des délais d'attente avant le traitement des clients
➢ Plus difficile à concevoir et à réaliser

Serveur avec connexion

Services de TCP

- Communication point-à-point
- Etablissement fiable de connexion
  - Le client est immédiatement averti si l'établissement échoue
- Remise fiable des données
- Contrôle de flux
- Transmission flux duplex
- Service de flots d'octets
  - TCP ne respecte pas la séparation de données en messages

Serveur avec connexion

- L'application n'a pas besoin de s'occuper de paquets perdus, …
- Nécessite un socket par client
  - La gestion de plusieurs connexions en parallèle peut être difficile
- Susceptible à des problèmes dus aux pannes des clients
  - Connexions 'zombie' si le client ne ferme pas correctement la connexion

Serveur sans connexion

Service d'UDP

- Communication multipoint-à-multipoint
- Remise non fiable des données
- Sans contrôle de flux
- Service de transmission de message

Serveur sans connexion

- Doit gérer la transmission fiable des données au niveau de l'application
  - Tester l'application dans des condition WAN, avec pertes, délais variables, …
- Plus résistant contre les problèmes dus au mauvais comportement des clients

Serveur itératif sans connexion

- Typiquement utilisé pour des applications simples requête – réponse
  - DNS, DHCP, …

Algorithme

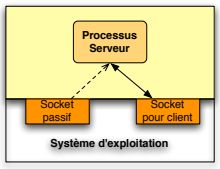
1. Créer un socket du type datagramme (*socket*)
2. Lier le socket à un port local (*bind*)
3. Boucle infinie
   a. Recevoir un message sur le socket (*recvfrom*)
   b. Construire la réponse
   c. Envoyer la réponse sur le socket (*sendto*)

Serveur itératif avec connexion

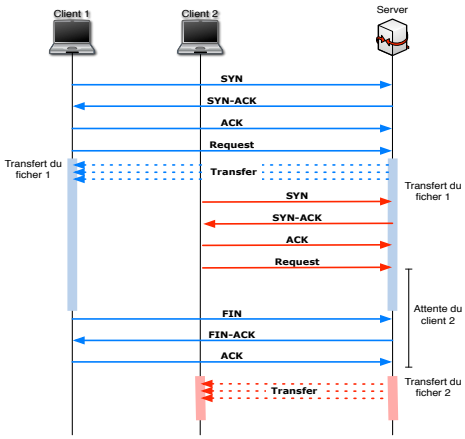
- Rare
- Utilisable uniquement pour des services ayant une charge faible
  - L'établissement d'une connexion peut prendre plusieurs secondes (perte de paquets)
  - Le traitement d'une tâche (p.ex. transfert de fichier) peut durer longtemps
    - Un deuxième client devra attendre

Algorithme

1. Créer un socket du type stream (*socket*).
2. Lier le socket à un port local (*bind*).
3. Mettre le socket en mode passif (*listen*).
4. Boucle infinie
   a. Accepter une nouvelle connexion (*accept*).
      Ceci crée un nouveau socket pour cette connexion.
   b. Lire toutes les données (*read*/*recv*) et envoyer les réponses (*send*/*write*).
   c. Fermer le socket de la connexion (*shutdown*/*close*).



Comportement du serveur itératif avec connexion



Programmation concurrente - Processus et Threads

Processus

- Instance d'un programme en cours d'exécution
- Possède son propre espace mémoire pour les données et le programme
- Complètement isolé des autres processus
- Communication interprocessus à l'aide de connexions TCP/UDP locales ou des tubes (*named pipes*)
- La création d'un nouveau processus est relativement lourde
  - Allocation de la mémoire, chargement du programme, …

Thread

- Un processus peut avoir plusieurs fils d'exécution
- Tous les threads d'un processus partagent le même espace mémoire
  - Facilite la communication entre les threads
  - Pose un problème lors de l'accès concurrent aux données
- La création d'un nouveau thread est plus rapide sur certaines plateformes (Windows)

Création d'un nouveau processus

- Appel système *fork()*
  - Crée une copie (presque) exacte du processus actuelle
  - L'OS exécute les deux processus en parallèle
    - Le deux processus continuent à exécuter le même programme après l'appel *fork()*
    - Le processus fils utilise une copie des données du processus père
    - Le processus fils hérite les sockets et fichiers ouverts du processus père
      - Permet aux deux processus d'accéder aux mêmes sockets et fichiers

Exemple de l'appel *fork()*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    int i;
    int sum = 0;

    fork();
    for (i=1; i<=5; i++) {
        printf("Value of i: %d\n", i);
        sum += i;
    }
    printf("The sum is %d\n", sum);
    exit(0);
}
```

Exécution

```
$ ./simple-fork
Value of i: 1
Value of i: 2
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
The sum is 15
Value of i: 3
Value of i: 4
Value of i: 5
The sum is 15
```

Faire diverger les deux processus

- Les deux processus continuent à exécuter le même programme
- L'appel *fork()* revoie comme résultat
  - au processus fils: une valeur 0
  - au processus père: l'identificateur du processus fils (PID > 0)
- Un test après *fork()* permet au processus de savoir s'il est le père ou le fils

Exemple

```
// Program 'diverge-fork'
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid;

    pid = fork();
    if (pid == 0) {
        printf("I am the child process!\n");
    } else {
        printf("I am the parent process! Child is %d.\n",pid);
    }
}
```

```
$ ./diverge-fork
I am the parent process! Child is 479.
I am the child process!
```

Exécuter un programme différent

- Dans l'exemple précédent les processus père et fils exécutent des parties différentes du même programme
- L'appel système *execve* permet de charger et exécuter un autre programme

  result = **execve**(*filename, argv, envp*)

- *filename*: nom du ficher avec le programme (binaire ou script)
- *argv*: vecteur d'arguments de ligne de commande
- *envp*: vecteurs de variables d'environnement, comme « CC=gcc »

- Typiquement utilisé seulement dans des serveurs multi-service
  - Serveur père reçoit les demandes pour plusieurs services et crée les processus fils en fonction du service demandé

Gestion de processus fils

- Un processus fils ne peut pas terminer complètement sans l'aide du processus père
- Processus « zombie » :
  - Processus mort qui est encore présent en mémoire
  - Permet au processus père d'obtenir des informations sur le processus fils

Exemple

```
~$ ./concurrent-tcp-server &
[1] 4989
Created new child process
  Closing child process
Created new child process
  Closing child process
Created new child process
  Closing child process

~$ ps
PID TTY          TIME CMD
4659 pts/0    00:00:00 bash
4989 pts/0    00:00:00 concurrent-tcp-server
4991 pts/0    00:00:00 concurrent-tcp-server <defunct>
4993 pts/0    00:00:00 concurrent-tcp-server <defunct>
4995 pts/0    00:00:00 concurrent-tcp-server <defunct>
4996 pts/0    00:00:00 ps
```

Gestion de processus fils

```c
// Program 'diverge-fork-wait'
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

void freeChild(int sig);

int main() {
    int pid;

    signal(SIGCHLD, freeChild); // Install signal handler

    pid = fork();
    if (pid == 0) {
        printf("I am the child process!\n");
        printf("Child is exiting now.\n");
        exit(0);
    } else {
        printf("I am the parent process! Child is: %d.\n", pid);
        sleep(5); // Wait 5 seconds before exiting.
        printf("Parent is exiting now.\n");
    }
}

// Signal handler
void freeChild(int sig) {
    pid_t pid;
    int stat;

    printf("*** Signal handler called.\n");
    while ((pid=waitpid(-1, &stat, WNOHANG)) > 0)
        printf("*** Child terminated: pid %d.\n", pid);
    return;
}
```


Gestion de processus fils

- Quand le processus fils termine, le processus père reçoit un signal *SIGCHLD*

Installer un gestionnaire du signal:

signal(*signal*, *function*)

- *signal*: SIGCHLD, SIGHUP, …
- *function*: int fun(int sig): Sera appelée quand le signal est reçu

Permettre à un processus fils de mourir

result = **waitpid**(pid, stat, option)

- *pid*: ID du processus fils attendu ou -1 pour tous les processus
- *stat*: Modifié par la fonction. Indique la raison de la terminaison du processus fils
- *option*: typiquement WNOHANG pour un appel non bloquant

– Retourne l'ID du processus mort

Serveur concurrent sans connexion

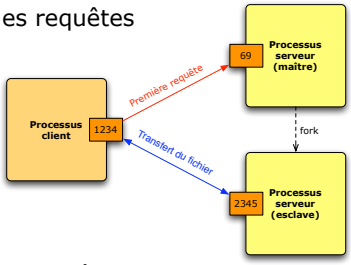
- Le processus maître attend les requêtes sur un socket DGRAM
- Pour le traitement de chaque requête un processus esclave est créé
- Rarement utilisé
  - Exemples: Serveur streaming multimédia

Processus maître
1. Créer un socket du type datagramme (*socket*).
2. Lier le socket à un port local (*bind*).
3. Boucle infinie
   a. Lire une nouvelle requête (*readfrom*).
   b. Créer un nouveau processus esclave (*fork*).

Processus esclave
1. Examiner la requête (copie de la mémoire).
2. Formuler la réponse.
3. Envoyer la réponse (*sendto*).
4. Terminer.

Gestion des sockets dans un serveur concurrent sans connexion

- Le processus fils hérite le socket du processus père
- Comment éviter des conflits entre les différents processus ?
  1. Si le processus fils ne fait qu'envoyer, mais ne lit jamais
     - Le processus fils peut utiliser le socket qu'il a hérité
     - Le processus père continue à lire les requêtes sur ce même socket
  2. Si le processus fils doit envoyer et recevoir
     - Le processus fils crée un nouveau socket sans connexion
     - Il communique avec le client sur ce socket
     - Le socket du client doit être non connecté



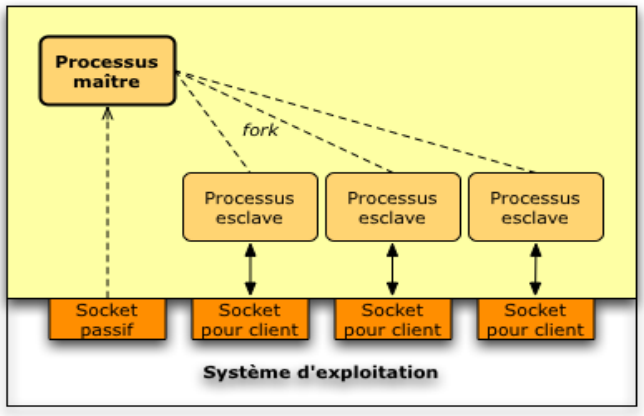
Serveur concurrent avec connexion

- L'architecture la plus courante
  - Serveurs Web, SSH, FTP, …
- Le processus maître attend des demandes de connexion sur un socket passif
- Lorsqu'une nouvelle connexion est acceptée un nouveau processus esclave est créé
- Le processus esclave hérite tous les sockets du processus maître
  - Le processus maître ferme le socket actif
  - Le processus esclave ferme le socket passif

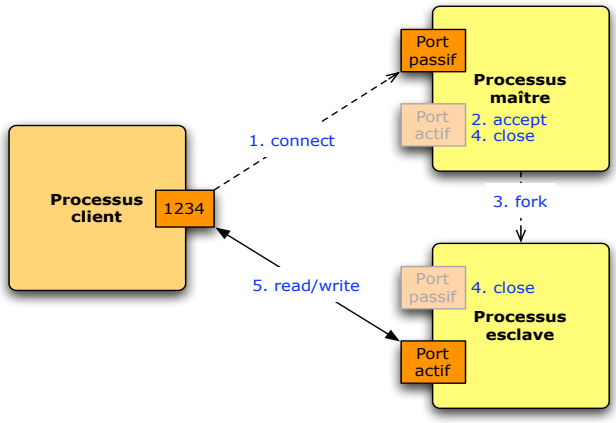
Processus maître
1. Créer un socket du type Stream (*socket*).
2. Lier le socket à un port local (*bind*).
3. Mettre le socket en mode passif (*listen*).
4. Boucle infinie
   a. Recevoir une nouvelle requête (*accept*).
   b. Créer un nouveau processus esclave (*fork*).

Processus esclave
1. Fermer le socket passif (*close*).
2. Utiliser le nouveau socket créé par *accept*.
3. Communiquer avec le client (*read/recv* et *write/send*).
4. Fermer le socket (*shutdown* et *close*).
5. Terminer.

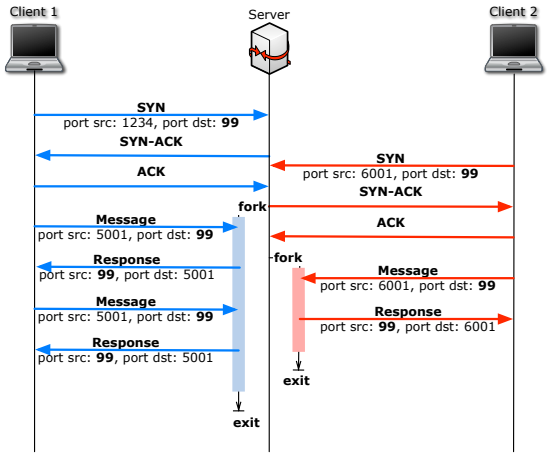
Gestion des sockets dans un serveur concurrent avec connexion



Comportement des processus d'un serveur concurrent avec connexion



Utilisation des numéros de port dans un serveur concurrent avec connexion



Serveur mono-tâche avec concurrence apparente

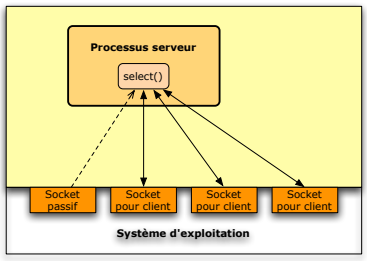
- Inconvénients des serveurs concurrents
  - La création d'un nouveau processus est lourde, notamment pour les connexions avec peu de trafic
  - La communication inter-processus est difficile

Serveur mono-tâche avec concurrence apparente

- Simule le traitement concurrent de plusieurs sockets
- Utilise la fonction *select* qui écoute plusieurs sockets à la fois
- La fonction *select* retourne quand un des sockets demande une action
- Le serveur peut effectuer l'action sans risque de se bloquer

Algorithme du serveur mono-tâche

1. Créer un socket passif (*socket, bind, listen*).
2. Ajouter le socket passif à la liste de sockets à gérer.
3. Boucle infinie
   a. Appeler *select* avec la liste des sockets à gérer.
   b. Si le socket à gérer est le socket passif :
      i. Accepter la connexion (*accept*).
      ii. Ajouter le nouveau socket à la liste des sockets à gérer.
   c. Si le socket à gérer est un socket connecté :
      i. Utiliser recv/read et send/write pour recevoir et envoyer.



La fonction *select*

int result = **select**(int *numdesc*,
fd_set **readfds*,
fd_set **writefds*,
fd_set **exceptfds*,
struct timeval *timeout*)

- Examine les descripteurs E/S fournis dans les arguments *readfds*, *writefds* et *exceptfds*
- Teste si une action de lecture, d'écriture ou de gestion d'exception est nécessaire
- Les objets surveillés peuvent être des fichiers, les flux E/S standard (*stdin*, *stdout*, *stderr*) et des sockets
- Select modifie les tableaux *readfds*, *writefds* et *exceptfds* pour indiquer quel descripteur demande une action
- Il est possible de fournir NULL à la place d'un tableau pour ne pas tester ce type d'action

Manipulation des tableaux de descripteurs

- Plusieurs macros permettent de manipuler les tableaux de descripteurs
  - *FD_ZERO(fdset)* : initialiser le tableau à zéro
  - *FD_SET(fd, &fdset)* : ajouter le descripteur fd
  - *FD_CLR(fd, &fdset)* : enlever le descripteur fd
  - *FD_ISSET(fd, &fdset)* : teste si fd est dans fdset
  - *FD_COPY(&fdset_orig, &fdset_copy)* : copie un tableau (sous BSD)
  - *memcpy(&fdset_copy, &fdset_orig, sizeof(fdset_copy))* : copie un tableau (sous Linux)

Autres paramètres de *select*

- Premier paramètre *numdesc*
  - Nombre maximum de descripteurs dans un des tableaux
  - Egal à *max(fd1, fd2, …)* + 1
- Paramètre *timeout*
  - Typiquement NULL
  - Permet de limiter le délai d'attente d'un événement
  - Type *struct timeval*

```
struct timeval {
    long tv_sec ;  /* secondes */
    long tv_usec ; /* microseconds */
} ;
```


Sockets prêts pour la lecture

- La fonction *select* signale qu'un socket est prêt pour la lecture dans ces conditions:
  - Données reçues :
    - Données peuvent être lues avec *read/recv* sans risque de blocage
  - Connexion fermée :
    - Select signale que la lecture est possible mais *read* lit 0 octets
  - Arrivée d'une demande de connexion :
    - Sur un socket passif
    - Le serveur peut appeler *accept* sans risque de se bloquer
  - Erreur du socket :
    - Select signale que la lecture est possible
    - L'appel de *read* renvoie une erreur (-1) et la variable *errno* indique la cause

Sockets prêts pour l'écriture

- La fonction *select* signale qu'un socket est prêt pour l'écriture dans ces conditions:
  - Ecriture est possible:
    - Données peuvent être écrites avec *write/send* sans risque de blocage
  - Erreur du socket :
    - *Select* signale que l'écriture est possible
    - L'appel de *write* renvoie une erreur (-1) et la variable *errno* indique la cause
  - Socket déjà fermé pour l'écriture
    - Ecriture donnera une erreur
    - Le programme doit donc faire attention de ne pas passer des sockets fermés à la fonction *select*

Traitement d'exceptions

- Le tableau *exceptfds* ne sert pas à traiter des erreurs de sockets
  - Des erreurs sont déjà signalées avec les tableaux *readfds* et *writefds*
- *Select* positionne un descripteur dans le tableau *exceptfds* à 1 si des données urgentes sont arrivées
- Données urgentes : données hors-bande de TCP, avec le fanion TCP URGENT

Exemple d'un serveur mono-tâche avec select

- Voir support de cours, p. 75

Performances des serveurs concurrent et mono-tâche

- Un serveur avec *select* est très performant
  - Le serveur concurrent multi-processus avec TCP accepte plusieurs dizaines de connexions par secondes
  - Le serveur mono-tâche avec select accepte une centaine de connexions par seconde
    - Sous Linux: peut gérer jusqu'à 1023 connexions, comme le nombre de descripteurs ouverts est limité
    - Mac OS X: jusqu'à 255 clients