

Programmation client/serveur avec les sockets

- déploiement du serveur et
gestion de clients multiples -

Module

Systemes Communicants et Synchronisés

Master Informatique
1ère année

L. Philippe & V. Felea

Introduction

- sur la base de la communication
 - ▶ support : mode connecté, non-connecté, autre...
 - ▶ utilisation du support
- mettre en place des applications : historique
 - ▶ simples : transfert fichiers, connexion distante
 - ▶ plus complexes : accès aux fichiers, accès au terminal, ...
 - ▶ intégrant le réseau : accès base distante, exécution de requêtes, ...
 - ▶ réparties : collaboratives, parallèles, ...

Client / Serveur

- définition : système logiciel

- ▶ fonctionnalité proposée, sous la forme d'un service, par un serveur à destination d'un client

- mise en œuvre

- ▶ programme = code + données
- ▶ découpage entre client et serveur
- ▶ définition des échanges : messages et synchronisation
- ▶ matériel : réseau, plusieurs machines

- intérêt

- ▶ gain attendu : modularité / souplesse
- ▶ risque : instabilité, sécurité, surcharge, etc.

Mise en œuvre

- difficultés

- ▶ réalisation des communications

- ◆ protocole d'échange

- langage d'échange

- codage

- ◆ synchronisations

- ◆ gestion de plusieurs requêtes simultanées

- ▶ répartition des données

- ◆ centralisées sur le serveur

- ◆ partagées client/serveur

Client / Serveur

- exemples

- ▶ serveur web (Apache Tomcat, Microsoft IIS – protocole HTTP)
- ▶ serveur de base de données (MySql, Microsoft SQL Server, Oracle DB – protocole 2PL)
- ▶ serveur d'impression (CUPS – protocoles LPD, IPP)
- ▶ serveur de fichiers (NFS – protocoles NFS, FTP)

- architecture de système

- ▶ J2EE
- ▶ Corba
- ▶ Web Services

Services

- serveur
 - ▶ numéro de port
 - ▶ déploiement
- client
 - ▶ adresse de machine - *gethostbyname*

Attribution d'un numéro de port (1)

- ▶ */etc/services* (Unix)

- ▶ *%SystemRoot%\System32\Driver\etc\services*
(Windows)

- IANA (Internet Assigned Numbers Authority - 1970)

« The IANA team is responsible for the operational aspects of coordinating the Internet's unique identifiers and maintaining the trust of the community to provide these services in an unbiased, responsible and effective manner »
(www.iana.org/about/)

- ▶ well-known services : 0 → 1023

- ▶ registered ports : 1024 → 49151

- ▶ dynamic/private ports : 49152 → 65535

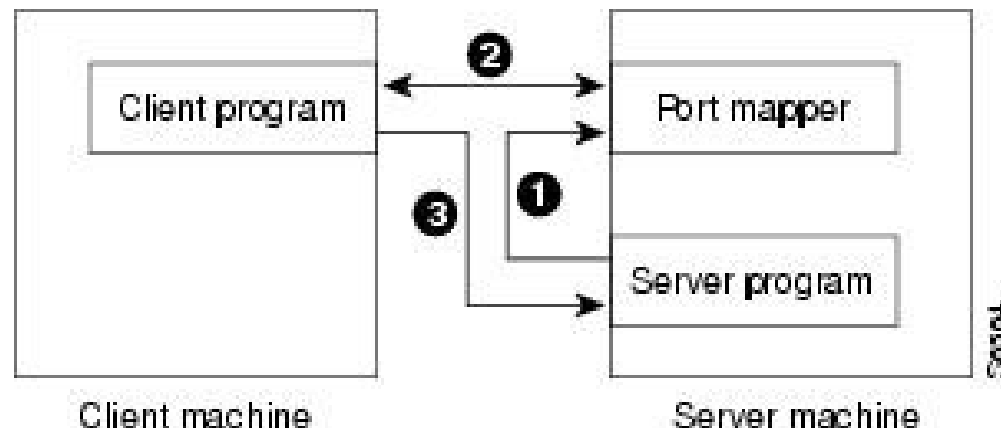
(procédure de réservation : [RFC4340], Section 19.9)

- *getservbyname* (structure *servent*)

- ▶ exemples : 22 – *ssh* 25 – *smtp* 80 – *http*

Attribution d'un numéro de port (2)

- gestion particulière : service NIS
 - ▶ Network Information Service : permet le partage des informations administratives (de configuration du système - nom d'hôte, utilisateur) sur le réseau
 - ▶ utilise RPC pour l'attribution des ports – attribution dynamique (*port mapper* – service *sunrpc* / port 111)



Déploiement du serveur

- manuel
 - ▶ dépendance du terminal de contrôle
 - ▶ → arrêté par signaux
- daemon : dissociation du terminal
- automatique

Dissociation du terminal de contrôle

- permet que le serveur ne soit plus tributaire du terminal qui le lance
 - ▶ *close* : 0, 1, 2 (entrée, sortie, erreur standards)
 - ▶ *open("/dev/null", O_RDWR)* : 0, 1, 2
 - ▶ *fd = open("/dev/tty", O_RDWR)* : terminal de contrôle
 - ▶ *ioctl(fd, TIOCNOTTY, 0)* : dissociation
 - ▶ *close(fd)*
- cas particulier : CTRL+C est ignoré

Déploiement automatique

- services sous Unix

- ▶ niveau d'exécution **N** : *runlevel* (dépend de la distribution)
- ▶ */etc/rc**N**.d* : liens vers */etc/init.d* (scripts)
- ▶ *S**NB**nom* : démarrage (start), ordre **NB**
- ▶ *K**NB**nom* : arrêt (kill), ordre **NB**

- service sous Windows

- ▶ *services.msc* / *msconfig* (services non Microsoft)
- ▶ types de démarrage : automatique, manuel, désactivé

Internet Daemon

- xinetd (eXtented InterNET Daemon)

- ▶ permet le lancement de daemons à la demande

- ▶ limite le nombre de daemons

- ▶ remplace inetd, plus sécurisé

- ▶ configuration

- ♦ /etc/xinetd.conf : fichier de configuration globale

- ♦ /etc/xinetd.d : répertoire contenant les fichiers de spécification des services

fichier *imap*

```
service imap
{
    socket_type      = stream
    protocol         = tcp
    wait             = no
    user             = root
    only_from        = 198.72.5.0 localhost
    server           = /usr/local/sbin/imapd
}
```

Types de serveurs

- selon la démarche de desservir les clients

- ▶ itératif

- ▶ concurrent

- selon le support

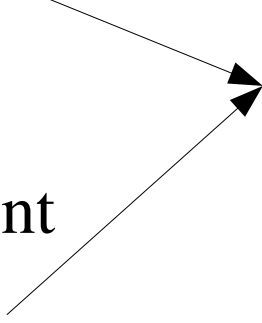
- ▶ connecté

- ▶ non connecté

- selon la gestion de l'information

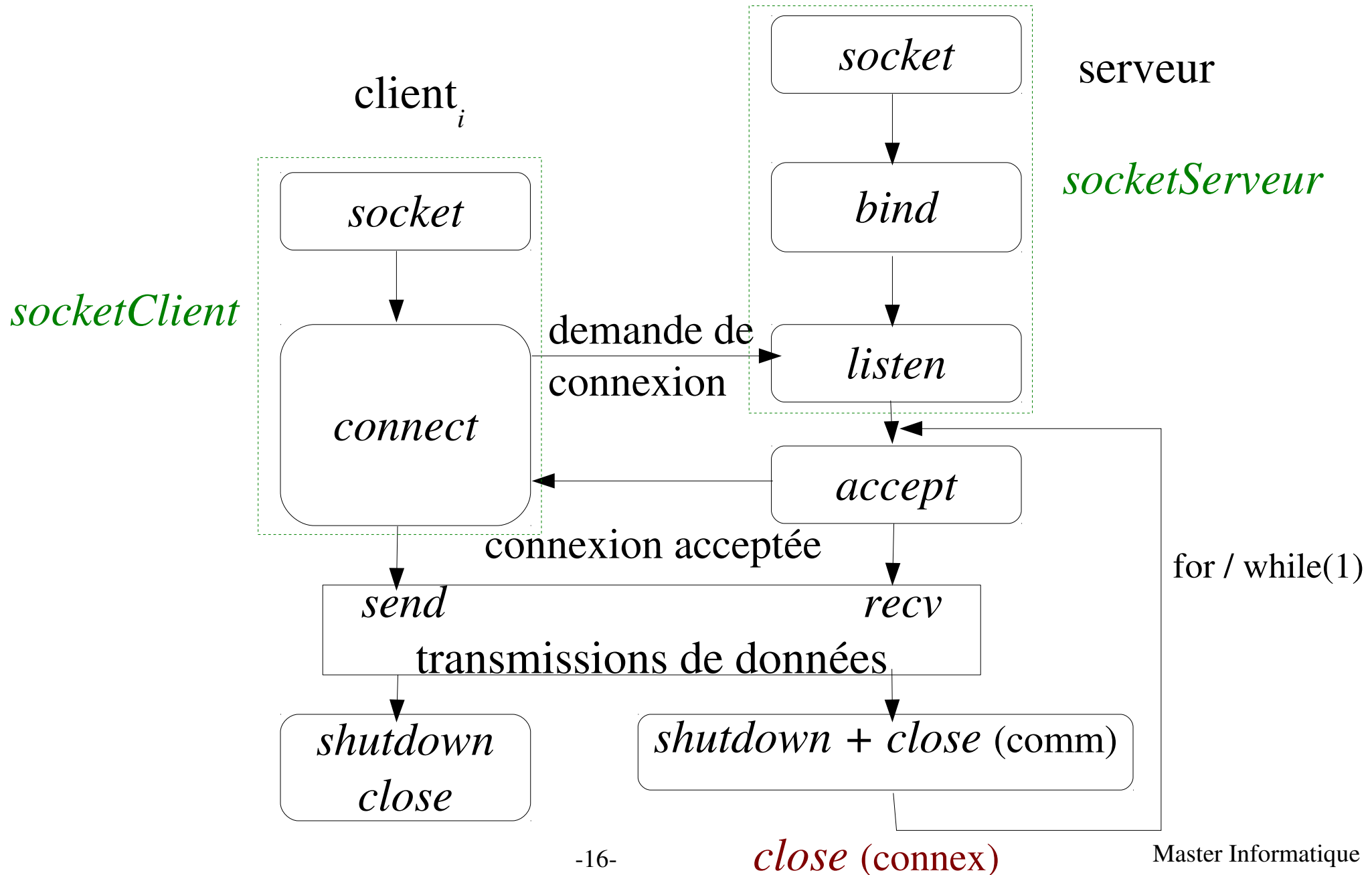
- ▶ sans état

- ▶ avec état



	mode connecté	mode non-connecté
itératif	serveur itératif en mode connecté	serveur itératif en mode non-connecté
concurrent	serveur concurrent en mode connecté	serveur concurrent en mode non-connecté

Serveur itératif – mode connecté



Structure générale du client en mode connecté

- ouverture connexion (*socketClient*)

- ▶ recherche de la machine destinatrice
(*gethostbyname/getaddrinfo*)

- ▶ création de la socket

- ▶ demande de connexion

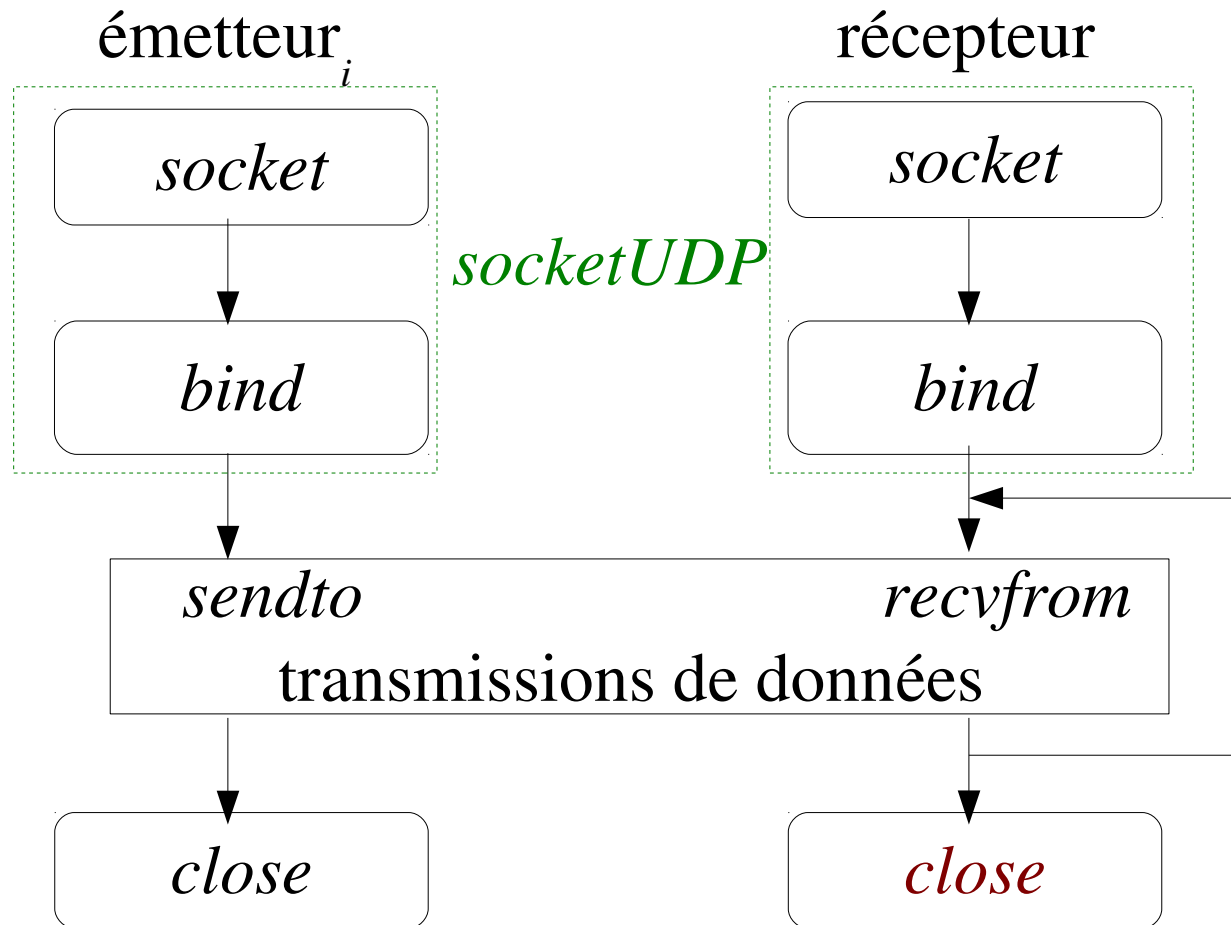
- accès au service (séquence de requête/réponse)

- fermeture de la connexion (flux de communication) et de la socket

```
...
int pid;
...
sockClient = socketClient (...);
...
for (...){ // while
    // composition requête requete
    // envoi requête
    err = send (sockClient, &requete, ...);
    if (err < 0) // erreur
        ...
    // réception réponse
    err = recv(sockClient, &reponse, ...);
    if (err <= 0) // erreur
        ...
}
shutdown(sockClient, 2);
close(sockClient);

exit(0);
```

Serveur itératif – mode non-connecté



Serveur itératif

- dessert les requêtes de manière itérative
- comportements différents entre mode connecté/mode non connecté (par rapport aux clients)
- simple à concevoir et implémenter
- adéquat quand le temps de réponse de la requête est relativement court (ex. service temps)
- non adéquat pour les requêtes de longue durée ou nécessitant des I/O

Serveur itératif – types support

- différence
- solution : select ou socket non bloquantes (pour le mode connecté)

Serveur concurrent

- problème

- ▶ *select* permet de

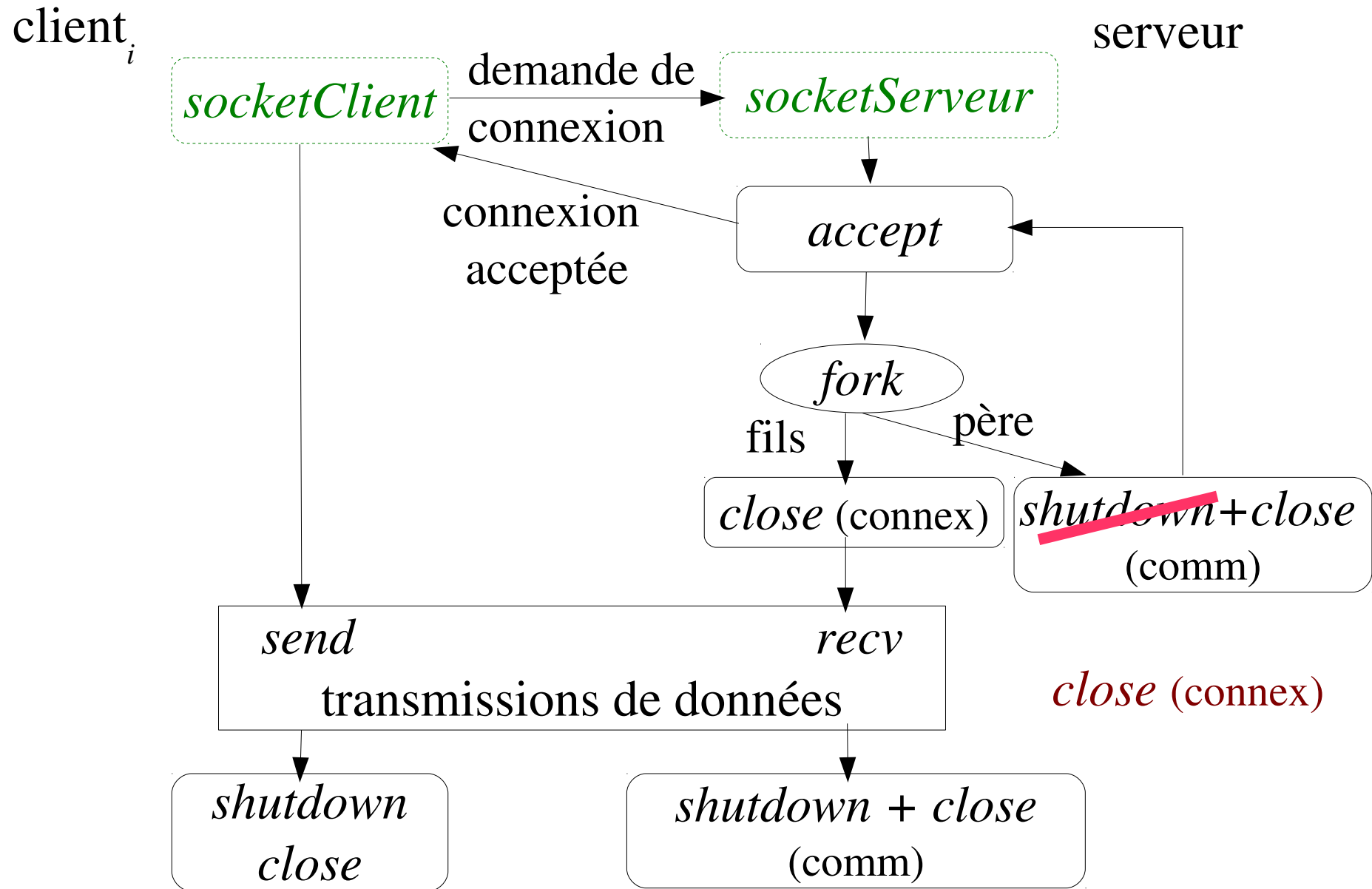
- ◆ traiter la première requête reçue
 - ◆ gagner le temps d'attente
 - ◆ traiter simultanément les clients → ne pas attendre la fin de l'un pour servir l'autre

- ▶ requêtes traitées séquentiellement

- ▶ besoin de traitement simultané de plusieurs requêtes provenant des clients différents

- solution : création de nouveaux processus (concurrents)

Serveur concurrent – mode connecté (1)



Serveur concurrent – mode connecté (2)

- structure générale du serveur

- ▶ processus père

- ◆ fork
 - ◆ ferme la socket de communication

- ▶ processus fils

- ◆ ferme socket de connexion
 - ◆ traite les requêtes client
 - ◆ termine

- exemple : xinetd

```
...
int pid;
...
sockCont = socketServeur (...);
...
while (1) {
    ...
    sockTrans = accept(sockCont, ...);
    pid = fork();
    switch (pid) {
        case 0 :    // processus fils
                    close(sockCont);

                    ...
                    procRequest(sockTrans);
                    exit(0);
        case -1 : // erreur
                    perror("serveur : erreur fork");
                    break;
        default : // processus père
                    shutdown(sockTrans),
                    close(sockTrans);
    }
}
```

Serveur concurrent – mode non-connecté (1)

- un processus par réception

(mode connecté : un processus par client)

- ▶ concurrence : traitement de la requête
- ▶ problèmes : création, suivi des requêtes de l'émetteur

- un processus par émetteur

- ▶ exemple : TFTP (UDP, port 69)
- ▶ concurrence : réception et traitement des requêtes d'un émetteur

Serveur concurrent – mode non-connecté (2)

- un processus par émetteur : création de nouvelles sockets
 - ▶ père
 - ▶ pas de demande de connexion → message émetteur
 - ▶ pb : informer l'émetteur
 - ◆ nouveau numéro de port
 - ◆ adresse de l'émetteur
 - ◆ gestion : ajouts dans l'application

Serveur concurrent – mode non-connecté (3)

● structure générale du serveur

▶ processus père

- ♦ réception d'une requête
- ♦ création nouvelle socket
- ♦ envoi le nouveau numéro de port
- ♦ fork
- ♦ fermeture de la socket

▶ processus fils

- ♦ fermeture socket initiale
- ♦ traitement des requêtes émetteur
- ♦ terminaison

```
...
int pid, portEmet;
...
sockPere = socketUDP(...);
...
while (1) {
    ...
    err = recvfrom(sockPere, ..., &addrEmet, ...);
    sockFils = socketUDP(portEmet);
    err = sendto(sockPere, &portEmet, addrEmet, ..);
    pid = fork();
    switch (pid) {
        case 0 : // processus fils
            close(sockPere);
            ...
            procRequest(sockFils);
            exit(0);
        case -1 : // erreur
            perror("serveur : erreur fork");
            break;
        default : // processus père
            close(sockFils);
            portEmet++;
    }
}
```


Modèles de concurrence chez le serveur (1)

- terminaison des processus fils sous Unix/Linux
 - ▶ à sa mort, un processus rend un code (*status*) à son père
 - ▶ dans le père : obtenu par *wait(int *status)*
 - ▶ tant que le père n'a pas reçu le *status* le processus n'est pas supprimé
 - ▶ mort en attente = zombie (*ps* → *defunct*)
 - ▶ le père doit faire *wait*, mais bloquant
 - ◆ sauf *waitpid(-1, &status, WNOHANG)*
 - ▶ le fils envoie un signal *SIGCHLD*, ignoré par défaut

Modèles de concurrence chez le serveur (2)

- solution : mettre en place un gestionnaire de signal (*handler*)

```
#include <signal.h>
```

```
typedef void (*sighandler_t)(int)
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

```
#include <signal.h>
void finFils(int sig) {
    int status;
    // wait for the child to be finished
    wait(&status);
    ...
}

main(int argc, char** argv) {
    ...
    // positionnement du signal
    signal(SIGCHLD, finFils);
    ...
}
```

Modèles de concurrence chez le serveur (3)

- solution : mettre en place un gestionnaire de signal (*handler*)
- solution 2 - sigaction

```
#include <signal.h>
```

```
typedef void (*sighandler_t)(int)
```

```
int sigaction(int signum, const struct sigaction *act,  
              struct sigaction *oldact)
```

```
#include <signal.h>
void sigchld_hdl (int sig){ ... }

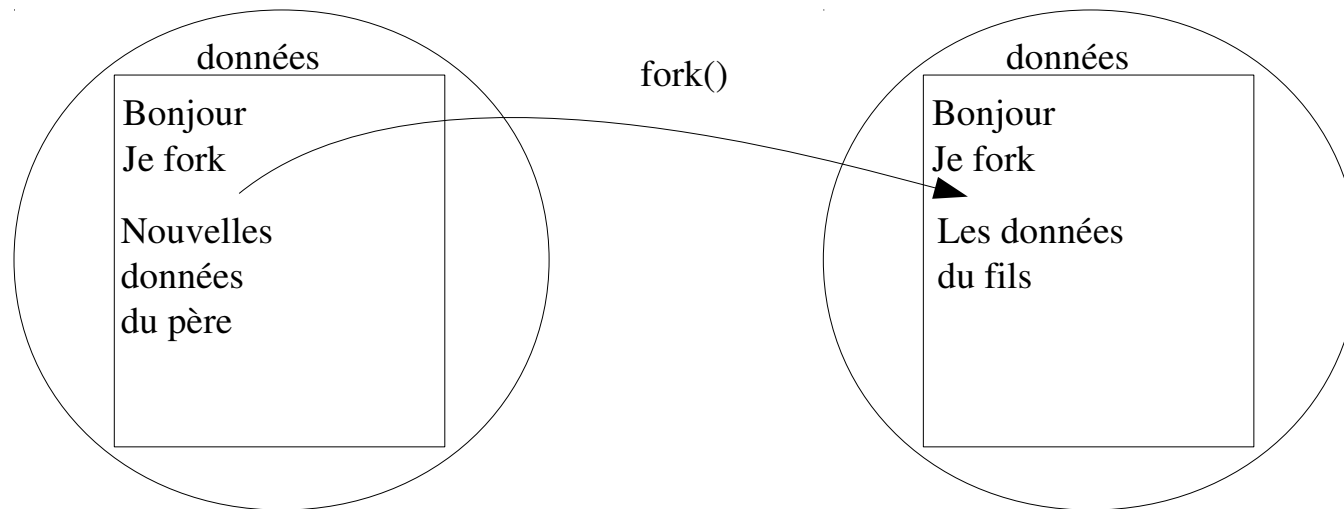
// main
    struct sigaction act;
    memset (&act, 0, sizeof(act));
    act.sa_handler = sigchld_hdl;
    if (sigaction(SIGCHLD, &act, 0))
        // error
```

```
struct sigaction {
    void    (*sa_handler)(int);
    void    (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int     sa_flags;
    void    (*sa_restorer)(void);
};
```

Modèles de concurrence chez le serveur (3)

- partage de données

- ▶ chaque processus dispose de ses propres données
- ▶ père/fils : recopie = données différentes



- ▶ pas de mémorisation de données client

Modèles de concurrence chez le serveur (4)

- utilisation des threads (processus légers)
 - ▶ plusieurs exécutions dans 1 contexte mémoire
 - ▶ partage de données
- pool de processus : création / destruction de processus
 - ▶ ensemble de processus pré-alloués
 - ▶ attente collective sur la socket de connexion
 - ▶ avantage
 - ◆ coût de la création dynamique d'un processus
 - ▶ inconvénient
 - ◆ exploitation des ressources (bon compromis disponibilité – ressources impliquées)

Serveur concurrent (1)

- dessert plusieurs clients simultanément
- concurrence
 - ▶ soit traitement réellement simultané (utilisant plusieurs processeurs)
 - ▶ soit traitement simultané apparent (temps partagé)
- difficile à implémenter
- temps de réponse réduit quand les clients demandent des requêtes de durées variées, ou beaucoup I/O

Serveur concurrent (2)

- architectures plus complexes
 - ▶ service de boîte aux lettres
 - ▶ création d'instances de boîtes aux lettres
 - ▶ redirection car pas toujours connecté
- plusieurs processus
- plusieurs ports