Communication par *sockets*

Renaud Lachaize

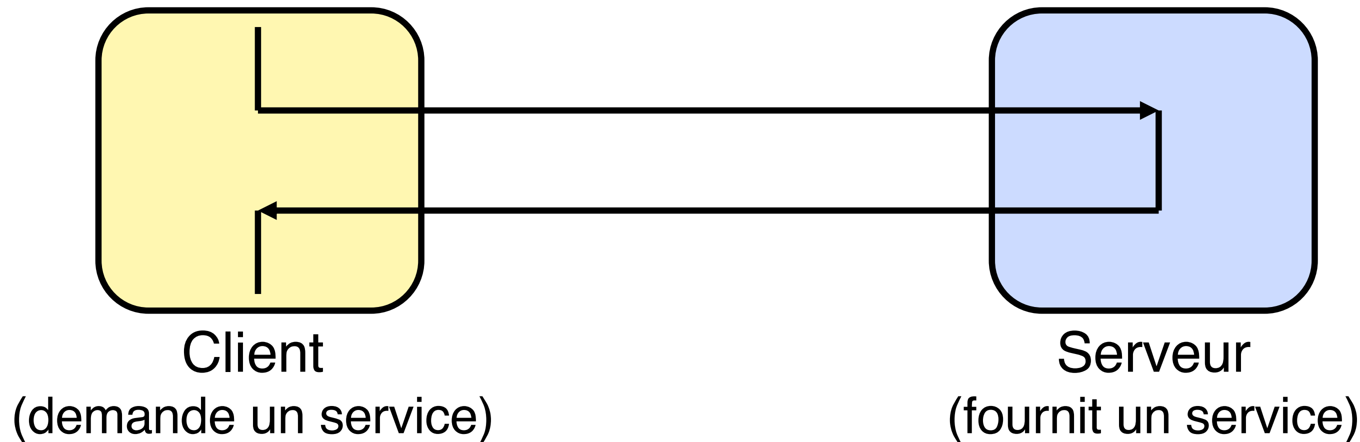
Université Grenoble Alpes

Renaud.lachaize @ imag.fr

Janvier 2017

Ce cours est basé sur les transparents de Sacha Krakowiak

Rappel : le réseau vu de l’utilisateur (1)

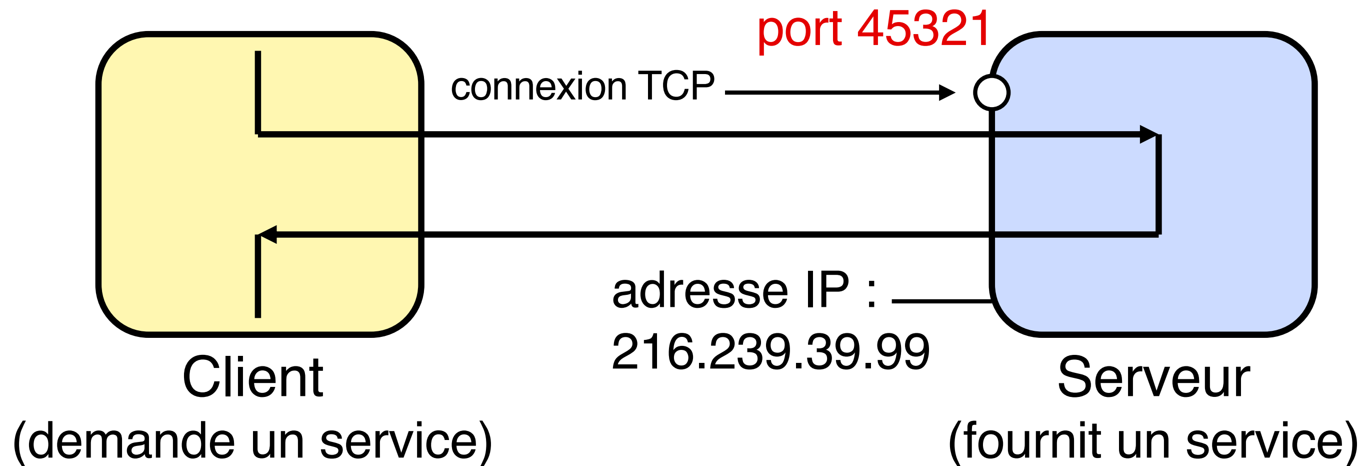


Le schéma client-serveur a été vu en TD pour des processus sur une même machine. Ce schéma se transpose à un réseau, où les processus client et serveur sont sur des machines différentes.

Pour le client, un service est souvent désigné par un nom symbolique.
Ce nom doit être converti en une adresse interprétable par les protocoles du réseau.

La conversion d’un nom symbolique (par ex. `www.google.com`) en une adresse IP (216.239.39.99) est à la charge du service DNS

Le réseau vu de l'utilisateur (2)



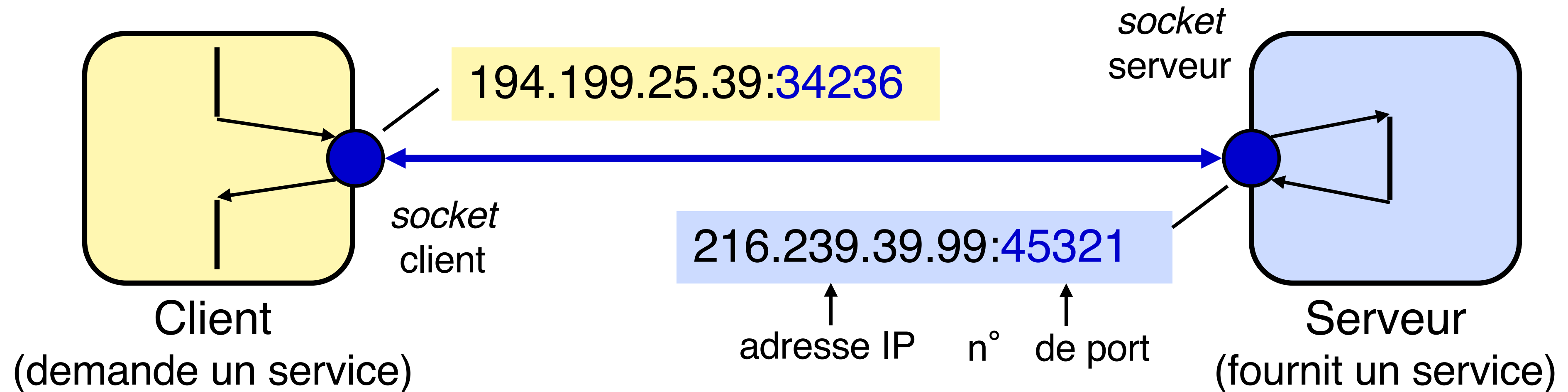
En fait, l'adresse IP du serveur ne suffit pas, car le serveur (machine physique) peut comporter différents services; il faut préciser le service demandé au moyen d'un numéro de port, qui permet d'atteindre un processus particulier sur la machine serveur.

Un numéro de port comprend 16 bits (0 à 65 535) et est associé à un protocole de transport donné (le port TCP n° i et le port UDP n° i désignent des objets distincts).

Les numéros de 0 à 1023 sont réservés, par convention, à des services spécifiques. Exemples (avec TCP) :

7 : echo
22 : SSH
25 : SMTP (acheminement mail)
80 : HTTP (serveur web)
443 : HTTPS (HTTP sécurisé)
465 : SMTPS (SMTP sécurisé)

Le réseau vu de l'utilisateur (3)



Pour programmer une application client-serveur, il est commode d'utiliser les *sockets*. Les *sockets* fournissent une interface qui permet d'utiliser facilement les protocoles de transport tels que TCP et UDP.

Une *socket* est simplement un moyen de désigner l'extrémité d'un canal de communication bidirectionnel, côté client ou serveur, en l'associant à un port.

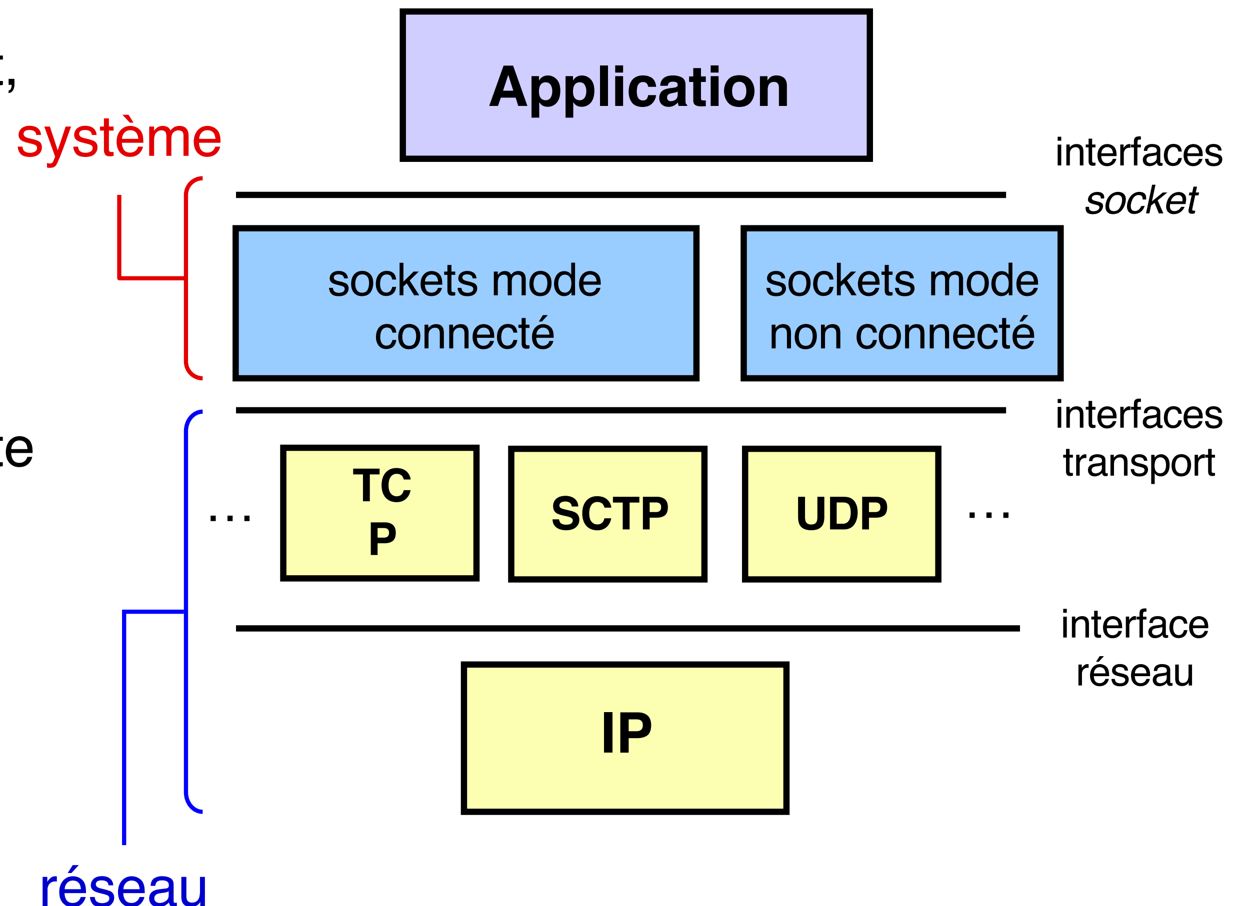
Une fois le canal de communication établi entre processus client et serveur, ceux-ci peuvent communiquer en utilisant les mêmes primitives (*read*, *write*) que pour l'accès aux fichiers/tubes.

Place des *sockets*

Les *sockets* fournissent une interface d'accès, à partir d'un hôte, aux interfaces de transport, notamment TCP et UDP

TCP (mode connecté) : une liaison est établie au préalable entre deux hôtes, et ensuite un flot d'octets est échangé sur cette liaison

UDP (mode non connecté) : aucune liaison n'est établie. Les messages sont échangés individuellement



Dans ce cours, ne considérons que des *sockets* TCP

Le protocole TCP

Principales caractéristiques de TCP

Communication bidirectionnelle par flots d'octets

Transmission fiable

Fiabilité garantie dès lors que la liaison physique existe

Transmission ordonnée

Ordre de réception identique à l'ordre d'émission

Contrôle de flux

Permet au récepteur de limiter le débit d'émission en fonction de ses capacités de réception

Contrôle de congestion

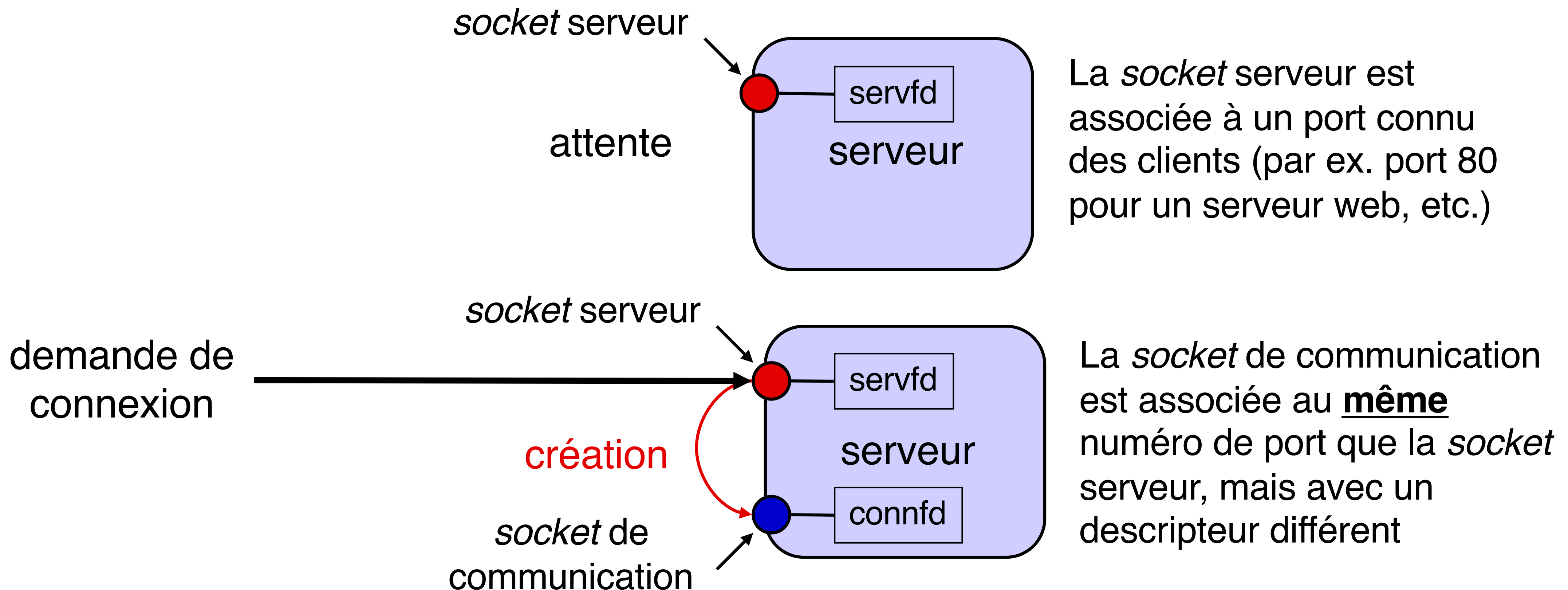
Permet d'agir sur le débit d'émission pour éviter la surcharge du réseau

Ne pas confondre contrôle de flux (entre récepteur et émetteur) et contrôle de congestion (entre réseau et émetteur)

Sockets côté serveur (1)

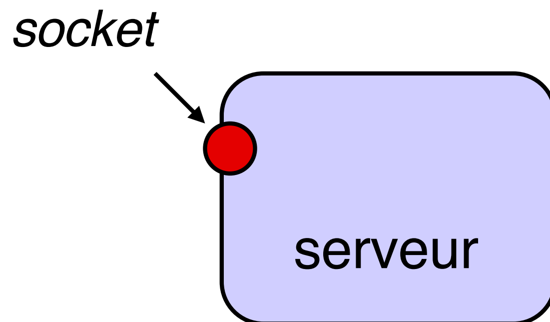
Un serveur en mode connecté doit attendre une nouvelle demande de connexion de la part d'un client, puis traiter la (ou les requêtes) envoyée(s) sur cette connexion par le client.

Les fonctions d'attente et de traitement sont séparées, pour permettre au serveur d'attendre de nouvelles demandes de connexion pendant qu'il traite des requêtes en cours.



Sockets côté serveur (2)

On procède en 4 étapes, décrites schématiquement ci-après (détails plus tard)

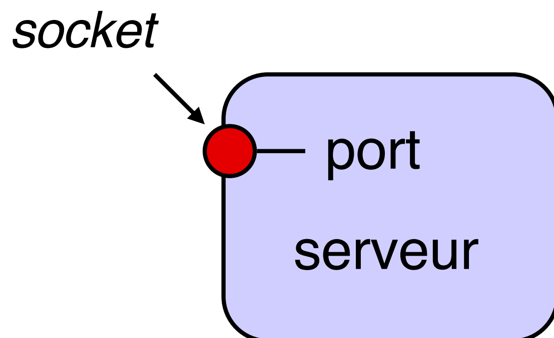


Étape 1 : créer une *socket* :

```
servfd = socket(AF_INET, SOCK_STREAM, 0);
```

AF_INET : Internet (IPv4) — SOCK_STREAM : TCP — 0 : protocole, sans usage ici

`servfd` est un descripteur analogue à un descripteur de fichier

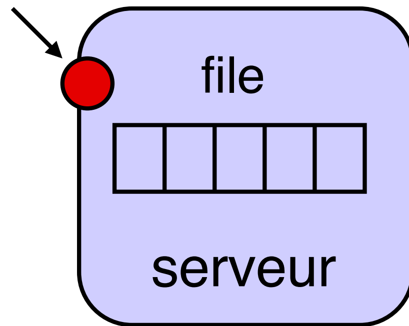


Étape 2 : associer la *socket* à un port (local) :

créer une structure `serveraddr` de type `sockaddr_in`

```
... /* explications plus loin ... */
 serveraddr.sin_port = port /* le n° de port choisi */
...
bind(servfd, &serveraddr, sizeof(serveraddr));
```

Sockets côté serveur (3)



Étape 3 : indiquer que c'est une *socket* serveur

```
#define QUEUE_SIZE 5   /* par exemple  */
listen(servfd, QUEUE_SIZE);
```

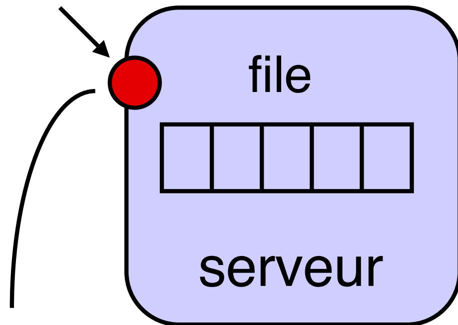
Taille max de la file d'attente des demandes

Une *socket* serveur est en attente de demandes de connexion.

Après un appel réussi à *listen*, le système d'exploitation de la machine serveur peut commencer à recevoir des demandes de connexion

Si une demande arrive pendant qu'une autre est en cours de traitement, elle est placée dans une file d'attente. Si une demande arrive alors que la file est pleine, elle est rejetée (pourra être refaite plus tard) ; voir primitive *connect* plus loin.

Sockets côté serveur (4)

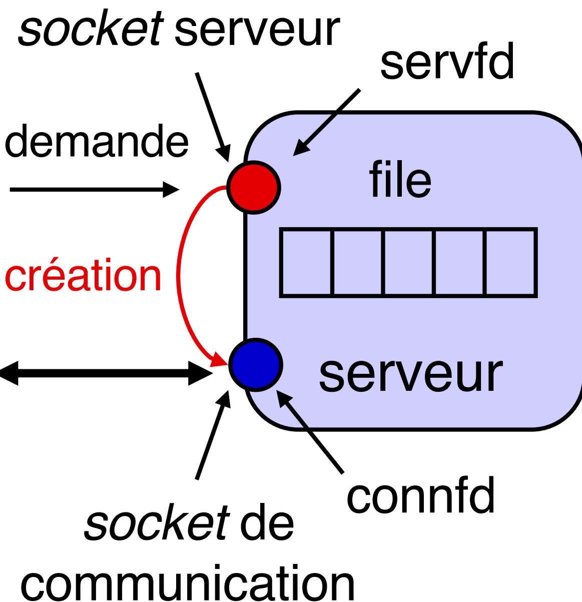


prête à accepter les demandes de connexion

Étape 4a : permettre à l'application de prendre connaissance des connexions

```
connfd = accept(servfd, &clientaddr, &addrlen);
```

la primitive `accept` est **bloquante**



Étape 4b : obtention d'un canal de communication par l'application (côté serveur)

```
connfd = accept(servfd, &clientaddr, &addrlen);
```

numéro de descripteur de la *socket* de communication qui a été créée

adresse et taille de la structure `sockaddr_in` du client dont la demande de connexion a été acceptée

Résumé des primitives pour les *sockets* côté serveur

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

int socket(int domain, int type, int protocol);
   /* crée une socket client ou serveur, renvoie descripteur) */

int bind(int sockfd, struct sockaddr *addr, int addrlen);
   /* associe une socket à une structure de description */

int listen(int sockfd, int maxqueuesize);
   /* déclare (et active) une socket comme serveur avec taille max queue   */

int accept(int sockfd, struct sockaddr *addr, int *addrlen);
   /* permet à un processus (applicatif) de prendre connaissance
      des nouvelles connexions de clients
      et d'obtenir un canal de communication pour chacune */
```

Il existe en outre une primitive `select` (non traitée ici - voir le livre CSAPP) et diverses variantes plus ou moins portables (`poll`, `epoll` sous Linux …)

Sockets côté client (1)

On procède en 2 étapes, décrites schématiquement ci-après

On suppose que l'on connaît l'adresse d'un serveur et le numéro de port d'une *socket* serveur sur celui-ci (un processus serveur est en attente sur ce port)

Étape 1 : créer une *socket* :

```
clientfd = socket(AF_INET, SOCK_STREAM, 0);
```


N.B. : opération identique à la création d'une *socket* serveur

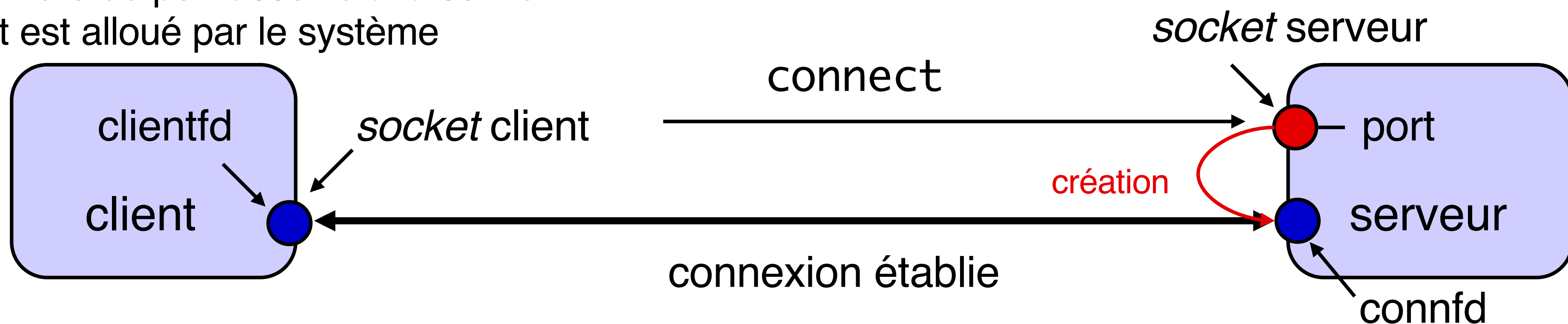
Le serveur est en attente sur la *socket* (`accept`)

Sockets côté client (2)

Étape 2 : établir une connexion entre la socket client et le serveur

```
struct sockaddr_in serveraddr;
...
  /* remplir serveraddr avec adresse et n°  port du serveur */

connect(clientfd, &serveraddr, sizeof(serveraddr));
  /* renvoie 0 si succès, 1 si échec */
```

`connect` envoie une demande de connexion vers la *socket* serveur



Le client et le serveur peuvent maintenant dialoguer sur la connexion

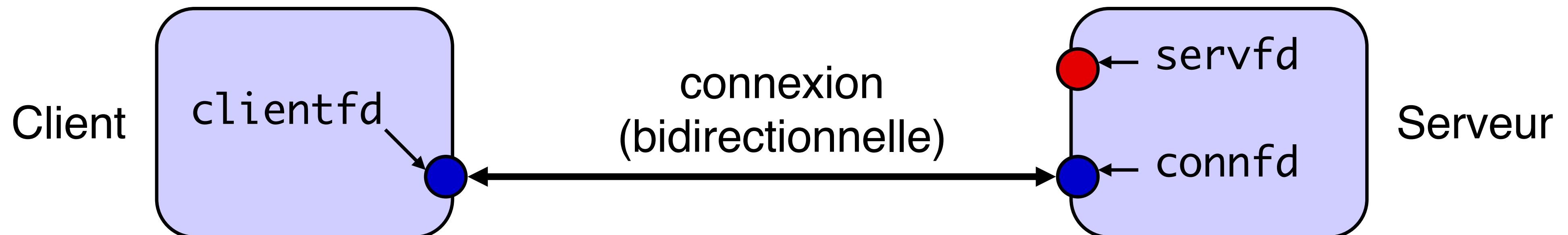
Résumé des primitives pour les *sockets* côté client

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

int socket(int domain, int type, int protocol);
   /* crée une socket client ou serveur, renvoie descripteur) */

int connect(int sockfd, struct sockaddr *addr, int addrlen);
   /* envoie une demande de connexion à un serveur              */
   /* serveur et numéro de port sont spécifiés dans sockaddr  */
   /* renvoie 0 si succès, -1 si échec                         */
```

```
struct sockaddr { /* structure générique pour sockets          */
   unsigned short sa_family;      /* famille de protocoles      */
   char           sa_data[14]; }  /* données d'adressage        */

struct sockaddr_in {/* structure pour sockets Internet          */
   unsigned short sin_family;     /* toujours AF_INET           */
   unsigned short sin_port;       /* numéro de port             */
   struct in_addr sin_addr;       /* adresse IPv4, ordre réseau */
   unsigned char  sin_zero[8]; }  /* remplissage pour sockaddr  */
```

Échanges sur une connexion entre *sockets*

Une fois la connexion établie, le client et le serveur disposent chacun d'un descripteur vers l'extrémité correspondante de la connexion.
Ce descripteur est analogue à un descripteur de fichier : on peut donc l'utiliser pour les opérations `read` et `write` ; on le ferme avec `close`.



Remarque 1. L'opération `lseek` (positionner un pointeur de lecture/écriture dans un fichier) n'a pas de sens pour les connexions sur *sockets*.

Remarque 2. Avec les *sockets*, il est vivement recommandé d'utiliser les primitives de la bibliothèque Rio (voir doc. technique) plutôt que `read` et `write`.

Identification des connexions

- Comment une machine peut-elle identifier la connexion réseau (et donc la socket concernée) lorsqu'elle reçoit un paquet ?

- En considérant l'adresse IP et le n° de port (TCP) de destination indiqués par l'émetteur ?
  - Mais plusieurs clients peuvent envoyer des paquets à destination d'un même couple <@ IP serveur, port serveur>
  - Donc ce n'est pas suffisant

- On peut identifier (de manière unique) la connexion associée à un paquet à partir du quadruplet :
  - <@ IP source, port source, @ IP destination, port destination>
  - Cette méthode est utilisée par le système d'exploitation pour aiguiller un paquet reçu vers la socket concernée

Echange de données à travers un réseau (1/2)
Protocole applicatif

- Un processus qui obtient des données à partir des sockets TCP manipule un flux d'octets

- C'est à l'application de découper ce flux de réception en messages

- Solutions
  - Messages de taille fixe
  - Messages de taille variable
    - En-tête de taille fixe indiquant le type/la taille du message complet
    - Marqueurs de fin de champs, de fin de message

Echange de données à travers un réseau (2/2) Interopérabilité

- Un protocole applicatif générique doit pouvoir fonctionner correctement entre machines hétérogènes
  - Processeurs avec des tailles de mots différentes
  - et/ou des « boutismes » (*endianness*) différents

- Implication n° 1 : spécification précise de la taille des types de base utilisés pour encoder/décoder un message
- Implication n° 2 : définition d'une convention de « boutisme/ordre réseau » (*network byte order*) pour les types de base dont la taille est supérieure à un octet
  - Émission : conversion entre « boutisme local » et « boutisme réseau » (voir les primitives `htons` et `htonl`)
  - Réception : opération inverse (voir les primitives `ntohs` et `ntohl`)
  - Cette précaution s'applique aussi aux adresses IP et numéros de port
  - Cette précaution n'est pas nécessaire pour les contenus basés sur des séries d'octets : chaînes de caractères, transferts de fichiers « opaques » ...

Une bibliothèque C pour les *sockets*

La programmation des *sockets* avec les primitives de base est complexe. Pour la simplifier, on a défini différentes bibliothèques qui fournissent une interface plus commode que celle des primitives de base.

Nous utiliserons la bibliothèque fournie dans `csapp.c`

Voir la description dans le Document Technique.

La bibliothèque comprend trois fonctions :

Côté serveur :

```
int Open_listenfd(int port)          /* encapsule socket, bind et listen */
        /* renvoie descripteur pour socket serveur, à utiliser dans Accept */
int Accept(int listenfd, struct sockaddr *addr, int *addrlen)
        /* identique à accept, avec capture des erreurs                  */
```

Côté client :

```
int Open_clientfd(char *hostname, int port) /* encapsule socket et connect */
        /* renvoie descripteur pour connexion côté client                */
```

Source : R. E. Bryant, D. O'Hallaron: *Computer Systems: a Programmer's Perspective*, Prentice Hall, 2003

Un application client-serveur avec *sockets* (1)

Principes de la programmation d'une application avec *sockets* (les déclarations sont omises).

Côté serveur :

```
/* crée une socket serveur associée au numéro de port "port"  */
listenfd = Open_listenfd(port)

while (TRUE) {
       /* accepte la connexion d'un client                   */
       /* on peut communiquer avec ce client sur connfd      */
       connfd = Accept(listenfd, &clientaddr, &clientlen);

       /* le serveur proprement dit      */
       /* qui lit requêtes et renvoie réponses sur connfd    */
       server_body(connfd);

       /* lorsque ce dialogue se termine, on ferme la connexion */
       Close(connfd);
       /* maintenant on va accepter la prochaine connexion   */
}
```

Un application client-serveur avec *sockets* (2)

Principes de la programmation d'une application avec *sockets* (les déclarations sont omises).

Côté client :

```
/* initialiser host et port (adresse serveur, numéro de port)  */

/* ouvrir une connexion vers le serveur, renvoie descripteur   */
clientfd = Open_clientfd(host, port);

/* envoyer requêtes et recevoir réponses sur clientfd         */
/* en utilisant read et write (ou plutôt bibliothèque Rio)    */
client_prog(clientfd);

/* à la fin des échanges, fermer le descripteur               */
Close(clientfd);
```

Un application client-serveur avec *sockets* (3)

Pour exécuter l'application :

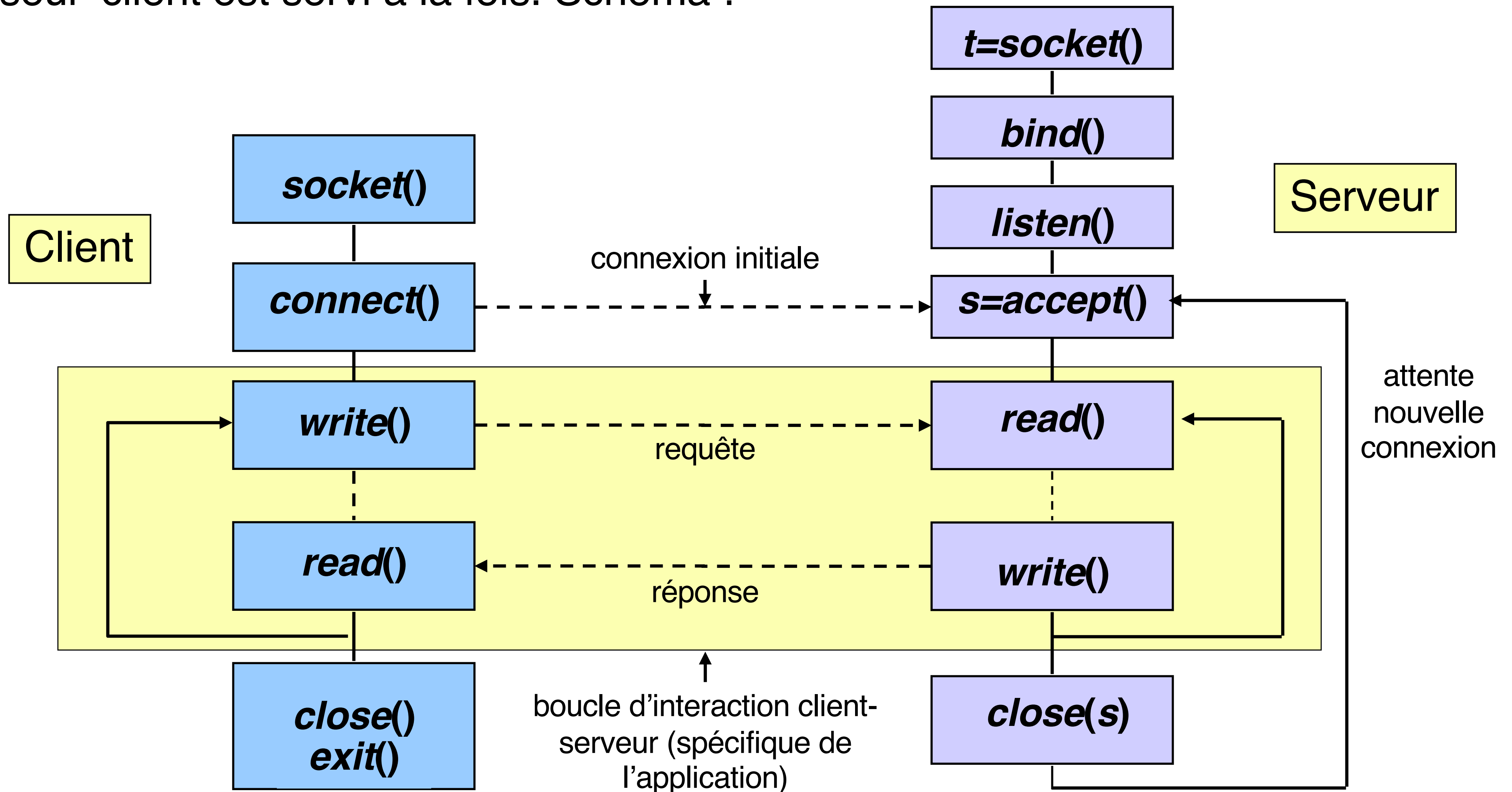
Lancer le programme serveur sur une machine, en indiquant un numéro de port (>1023, les numéros ≤1023 sont réservés) ; de préférence en travail de fond

Lancer le programme client sur une autre machine (ou dans un autre processus de la même machine), en spécifiant adresse du serveur et numéro de port

N.B. On n'a pas prévu d'arrêter le serveur (il faut tuer le processus qui l'exécute). Dans une application réelle, il faut prévoir une commande pour arrêter proprement le serveur

Client-serveur en mode itératif

Les programmes précédents réalisent un serveur en **mode itératif** : un seul client est servi à la fois. Schéma :



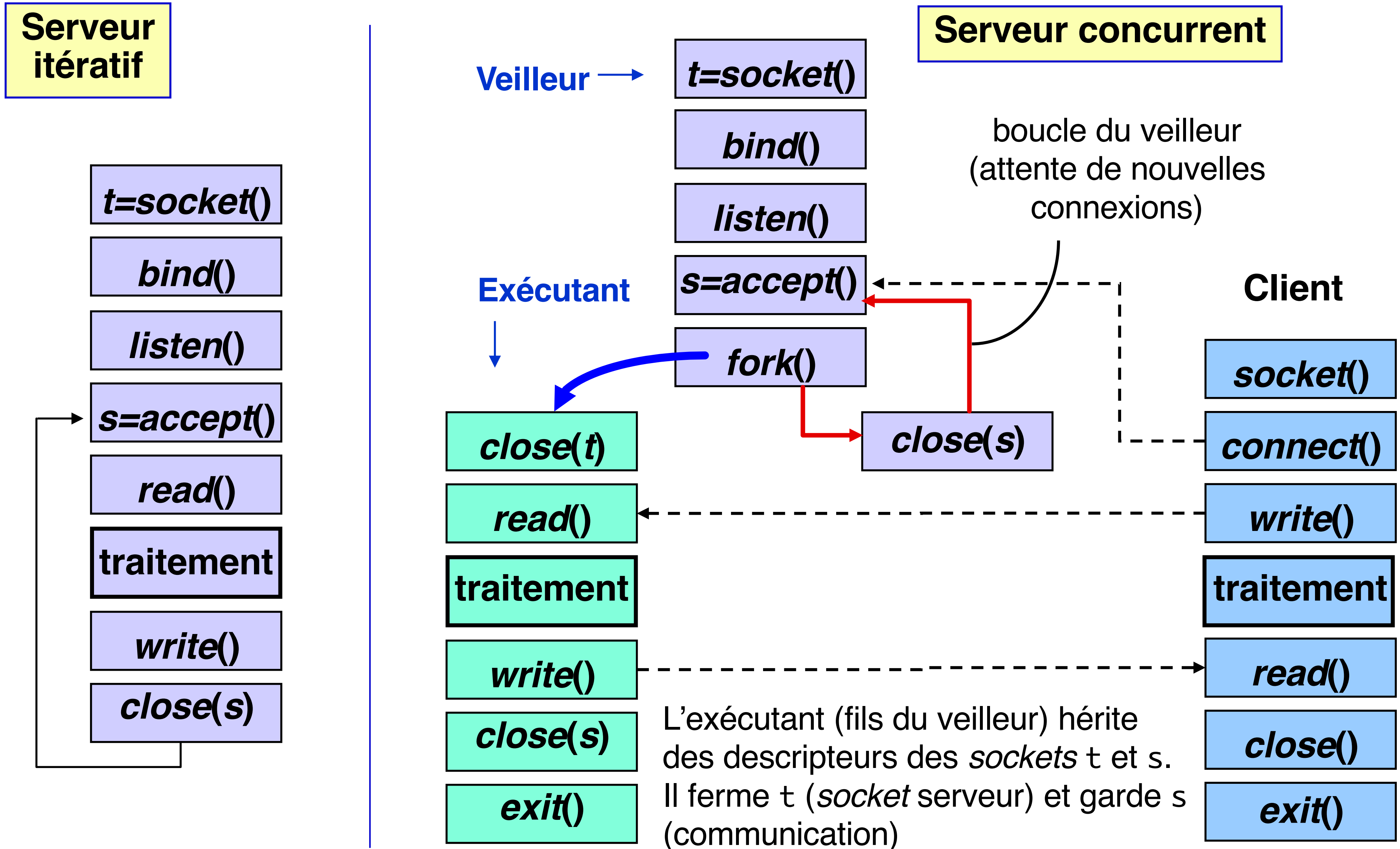
Client-serveur en mode concurrent (1)

Pour réaliser un serveur en **mode concurrent**, une solution consiste à créer un nouveau processus pour servir chaque demande de connexion, le programme principal du serveur ne faisant que la boucle d'attente sur les demandes de connexion.

Donc il y a un processus principal (appelé *veilleur*) qui attend sur `accept()`. Lorsqu'il reçoit une demande de connexion, il crée un processus fils (appelé *exécutant*) qui va interagir avec le client. Le veilleur revient se mettre en attente sur `accept()`. Plusieurs exécutants peuvent exister simultanément.

Il existe d'autres solutions (*threads*, multiplexage par `select/poll` …), non vues cette année.

Client-serveur en mode concurrent (2)



Exemple de client-serveur avec *sockets* (1)

Serveur echo : programme principal (appelle fonction echo)

```
int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;

    port = atoi(argv[1]); /* the server listens on a port passed
                              on the command line */
    listenfd = open_listenfd(port);

    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        printf("server connected to %s (%s)\n", hp->h_name, haddrp);
        echo(connfd);
        Close(connfd);
    }
}
```

SA : synonyme de `struct sockaddr`

boucle d'attente du serveur itératif

la fonction echo **prend ses requêtes et renvoie ses réponses sur** `connfd`**. Contient une boucle interne pour échanges client-serveur**

Source : R. E. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice Hall, 2003

Exemple de client-serveur avec *sockets* (2)

Serveur echo : la fonction echo

```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", n);
        Rio_writen(connfd, buf, n);
    }
}
```

lit une ligne et la range dans `buf`

boucle interne pour échanges client-serveur

renvoie la ligne (contenu de `buf`) au client)

La boucle se termine sur la condition EOF, détectée par la fonction RIO `Rio_readlineb`. Ce n'est pas un caractère, mais une condition (mise à vrai par la fermeture de la connexion par le client)

Source : R. E. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice Hall, 2003

Exemple de client-serveur avec *sockets* (3)

Client echo

```
#include "csapp.h"

/* usage: ./echoclient host port */
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];
    rio_t rio;

    host = argv[1];
    port = atoi(argv[2]);

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

- lit une ligne du clavier vers `buf` → `Fgets(buf, MAXLINE, stdin)`
- envoie le contenu de `buf` au serveur → `Rio_writen`
- reçoit dans `buf` , puis imprime, la réponse du serveur → `Rio_readlineb` / `Fputs`
- connexion au serveur (`host:port`) → `Open_clientfd`
- la boucle se termine par détection de EOF (*control-D*)

Source : R. E. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice Hall, 2003

Exemple de client-serveur avec *sockets* (4)

Mode d'emploi

sur le serveur

```
{imablade04$} server 4321 &
```

lance le serveur sur le port 4321

sur le client

```
{mandelbrot$} client imablade04.e.ujf-grenoble.fr 4321
```

appelle le serveur distant

Les programmes client et serveur sont indépendants, et compilés séparément. Le lien entre les deux est la connaissance par le client du nom du serveur et du numéro de port du service (et du protocole de transport utilisé).

Client et serveur peuvent s'exécuter sur deux machines différentes, ou sur la même machine. Le serveur doit être lancé **avant** le client.

Observer la liste des sockets TCP sur une machine

- Commande netstat
  - netstat -t (équivalent à : netstat -A inet --tcp)
- Options utiles
  - -a ou --all : permet d'afficher toutes les sockets existantes sur la machine (par défaut, seules les sockets connectées sont listées)
  - -l ou --listen : affiche uniquement les sockets serveurs
  - -p : affiche le pid du processus propriétaire d'une socket
  - -e : permet de connaître l'utilisateur associé au processus propriétaire d'une socket
  - --numeric-hosts : désactiver la résolution des noms de machines (affichage des adresses IP)
  - --numeric-ports : désactiver la résolution des numéros de ports (par défaut, les numéros de ports utilisés par les services usuels sont remplacés par le nom du service correspondant, à partir des informations disponibles dans le fichier /etc/services)

Observer la liste des socketsTCP sur une machine
Exemple

Configuration du test

- Serveur lancé sur la machine imablade04 (port 7777)
- Client lancé sur la machine mandelbrot

```
mandelbrot$ netstat -t -a --numeric-ports --numeric-hosts
Active Internet connections (servers and established)
Proto   Recv-Q   Send-Q   Local Address           Foreign Address        State
...
tcp     0        0        195.220.82.165:43103    195.220.82.164:7777    ESTABLISHED
...
```

```
imablade04$ netstat -t -a --numeric-ports --numeric-hosts
Active Internet connections (servers and established)
Proto   Recv-Q   Send-Q   Local Address           Foreign Address        State
...
tcp     0        0        0.0.0.0:7777            0.0.0.0:*              LISTEN
tcp     0        0        195.220.82.164:7777     195.220.82.165:43103   ESTABLISHED
...
```

Un autre outil pour l'étude des sockets : ss

- Commande
  - ss -t
- Options utiles
  - -a ou --all : permet d'afficher toutes les sockets existantes sur la machine (par défaut, seules les sockets connectées sont listées)
  - -l ou --listening : affiche uniquement les sockets serveurs
  - -p : affiche le pid du processus propriétaire d'une socket
  - -e : permet de connaître l'utilisateur associé au processus propriétaire d'une socket
  - -n ou --numeric : désactiver la résolution des noms de services
  - -r ou –resolve : activer la résolution des noms de machines et de services

Les primitives de la bibliothèque C : côté client

```
int open_clientfd(char *hostname, int port)
{
  int clientfd;
  struct hostent *hp;
  struct sockaddr_in serveraddr;

  if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1; /* check errno for cause of error */

  /* Fill in the server's IP address and port */
  if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
  bzero((char *) &serveraddr, sizeof(serveraddr));
  serveraddr.sin_family = AF_INET;
  bcopy((char *)hp->h_addr_list[0],
        (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
  serveraddr.sin_port = htons(port);

  /* Establish a connection with the server */
  if (connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr)) < 0)
    return -1;
  return clientfd;
}
```

Cette fonction ouvre une connexion depuis le client vers le serveur `hostname:port`

Erreur DNS : serveur pas trouvé

Remplir la structure `sockaddr_in` pour serveur

Conversion hôte -> réseau

descripteur de la *socket* de connexion côté client

Source : R. E. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice Hall, 2003

Les primitives de la bibliothèque C : côté serveur (1)

```
int open_listenfd(int port)
{
    int listenfd, optval=1;
    struct sockaddr_in serveraddr;

    /* Create a socket descriptor */
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    /* Eliminates "Address already in use" error from bind. */
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int)) < 0)
        return -1;

... (more)
```

Cette fonction crée une *socket* serveur associée au port `port` et met le serveur en attente sur cette *socket*.

Option pour permettre l'arrêt et le redémarrage immédiat du serveur, sans délai de garde (par défaut, il y a un délai d'environ 30 s., ce qui est peu commode pour la mise au point)

Source : R. E. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice Hall, 2003

Les primitives de la bibliothèque C : côté serveur (2)

Remplir la structure sockaddr_in **pour serveur**

```
...

  /* Listenfd will be an endpoint for all requests to port
       on any IP address for this host */
   bzero((char *) &serveraddr, sizeof(serveraddr));
   serveraddr.sin_family = AF_INET;
   serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
   serveraddr.sin_port = htons((unsigned short)port);
   if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
       return -1;

   /* Make it a listening socket ready to accept
      connection requests */
   if (listen(listenfd, LISTENQ) < 0)
       return -1;

  return listenfd;
}
```

Conversion hôte -> réseau

associe la *socket* avec le port port

déclare la *socket* comme *socket* serveur

renvoie un descripteur pour la *socket* serveur, prêt à l'emploi (avec accept**)**

Source : R. E. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice Hall, 2003