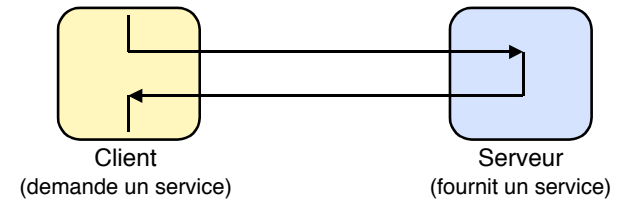


## Communication par sockets

**Sacha Krakowiak**  
Université Joseph Fourier  
Projet Sardes (INRIA et IMAG-LSR)  
<http://sardes.inrialpes.fr/people/krakowia>

## Rappel : le réseau vu de l'utilisateur (1)

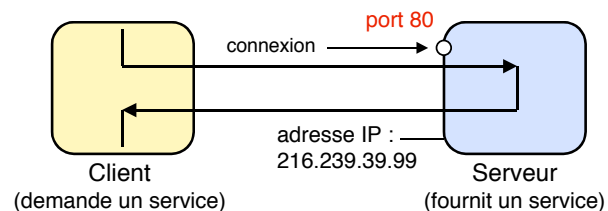


Le schéma **client-serveur** a été vu en TD pour des processus sur une même machine. Ce schéma se transpose à un réseau, où les processus client et serveur sont sur des machines différentes.

Pour le client, un service est souvent désigné par un nom symbolique (par exemple mail, `http://...`, telnet, etc.). Ce nom doit être converti en une adresse interprétable par les protocoles du réseau.

La conversion d'un nom symbolique (par ex. `http://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)

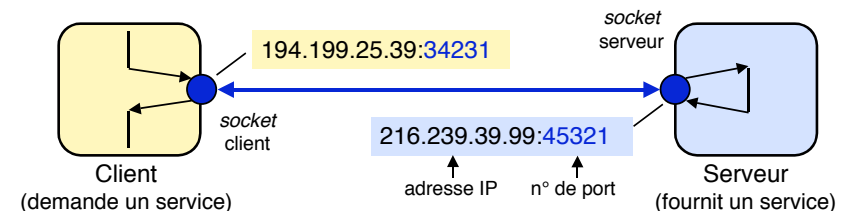


En fait, l'adresse IP du serveur ne suffit pas, car le serveur (machine physique) peut comporter différents services; il faut préciser le service demandé au moyen d'un **numéro de port**, qui permet d'atteindre un processus particulier sur la machine serveur.

Un numéro de port comprend 16 bits (0 à 65 535). Les numéros de 0 à 1023 sont réservés, par convention, à des services spécifiques. Exemples :

7 : echo	23 : telnet (connexion à distance)
80 : serveur web	25 : mail

## Le réseau vu de l'utilisateur (3)



Pour programmer une application client-serveur, il est commode d'utiliser les **sockets** (disponibles en particulier sous Unix). Les **sockets** fournissent une interface qui permet d'utiliser facilement les protocoles de transport TCP et UDP

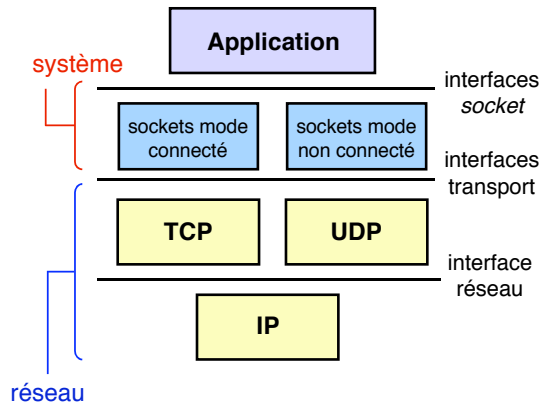
Une **socket** est simplement un moyen de désigner l'extrémité d'une connexion, côté émetteur ou récepteur, en l'associant à un port. Une fois la connexion (bidirectionnelle) établie via des **sockets** entre un processus client et un processus serveur, ceux-ci peuvent communiquer en utilisant les mêmes primitives (*read*, *write*) que pour l'accès aux fichiers.

## Place des sockets

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

**TCP (mode connecté)** : une liaison est établie au préalable entre deux hôtes, et ensuite les messages (plus exactement des flots d'octets) sont échangés sur cette liaison

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

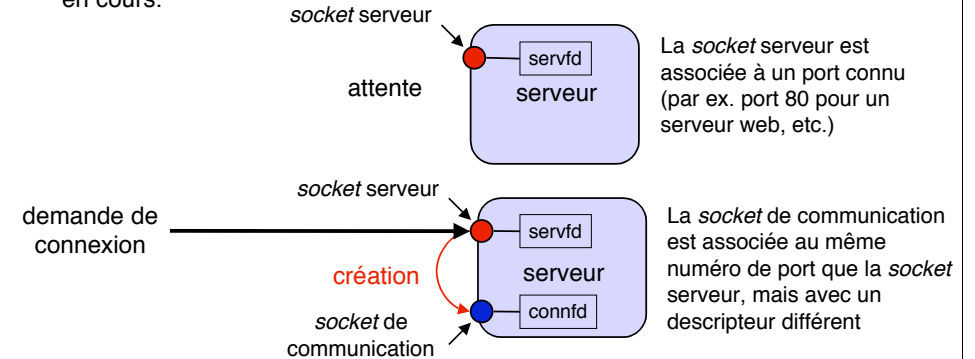


Nous ne considérons que des *sockets* en mode **connecté**

## Sockets côté serveur (1)

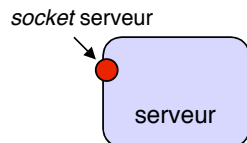
Un serveur fournit un service à des clients. Il doit donc **attendre** une demande, puis la **traiter**.

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



## Sockets côté serveur (2)

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

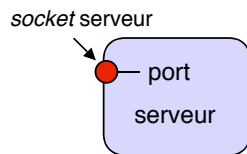


### Étape 1 : créer une socket :

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

↑ Internet    ↑ TCP

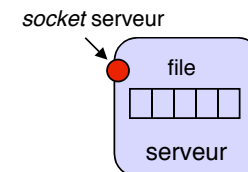
servfd est un descripteur analogue à un descripteur de fichier



### Étape 2 : associer la socket à un port :

```
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)



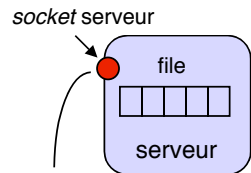
### Étape 3 : indiquer que c'est une socket serveur

```
#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. 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

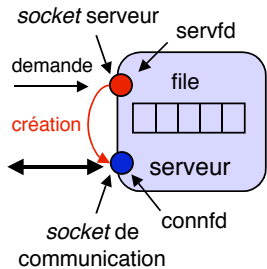
## Socket côté serveur (4)



**Étape 4a : se mettre en attente des demandes**  
`connfd = accept(servfd, &clientaddr, &addrlen);`

la primitive accept est **bloquante**

prête à accepter les demandes de connexion



**Étape 4b : après acceptation d'une demande**  
`connfd = accept(servfd, &clientaddr, &addrlen);`

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

↑  
 adresse et taille de la structure sockadr\_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 une socket comme serveur avec taille max queue */

int accept(int sockfd, struct sockaddr *addr, int *addrlen);
/* met une socket serveur en attente de demandes de connexion */
```

Il existe en outre une primitive select, non traitée ici

## Socket 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);`

↑  
 Internet    ↑  
 TCP

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



Le serveur est en attente sur la socket (accept)

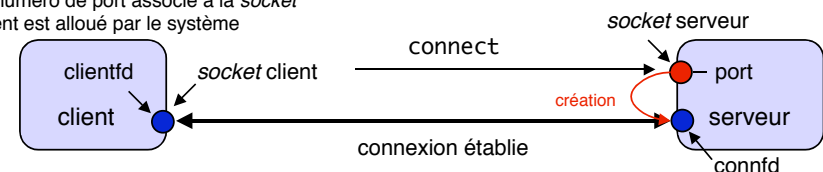
## Socket côté client (2)

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

```
struct sockaddr_in serveraddr;
...
/* remplir server addr 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 numéro de port associé à la socket client est alloué par le système



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>

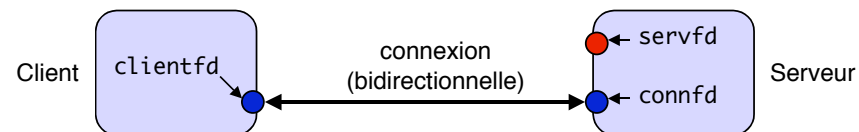
int socket(int domain, int type, int protocol);
/* crée une socket client ou serveur, renvoie descripteur) */

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 pour sockets génériques */
    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 IP, 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 document technique n°2, placard) plutôt que read et write.

## 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 n°3](#) (placard)

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 :

```
listenfd = Open_listenfd(port)
/* crée une socket serveur associée au numéro de port "port" */
while (TRUE) {
    connfd = Accept(listenfd, &clientaddr, &clientlen);
    /* accepte la connexion d'un client */
    /* on peut communiquer avec ce client sur connfd */
    server_body(connfd); /* le serveur proprement dit */
    /* qui lit requêtes et renvoie réponses sur connfd */
    /* lorsque ce serveur 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) */
clientfd = Open_clientfd(host, port)
/* ouvre une connexion vers le serveur, renvoie descripteur */

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

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

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

Pour exécuter l'application :

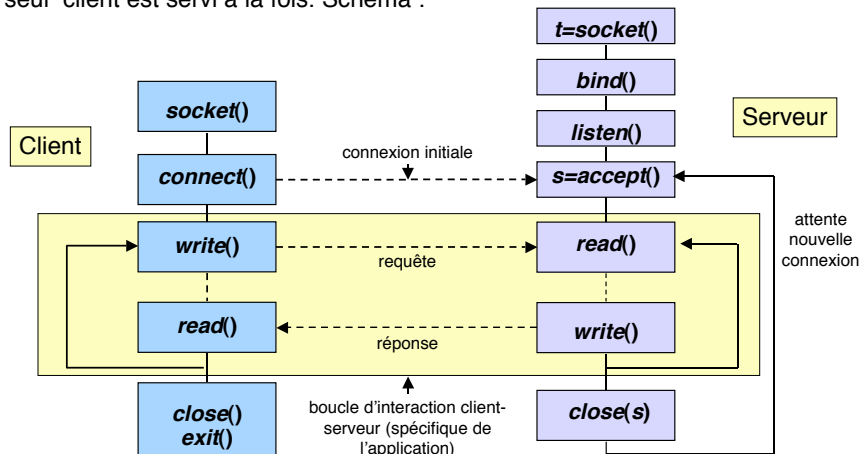
Lancer le programme serveur sur une machine, en indiquant un numéro de port (>1023, les numéros ≤1023 sont réservés) ; de préférence en travail de fond

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 :



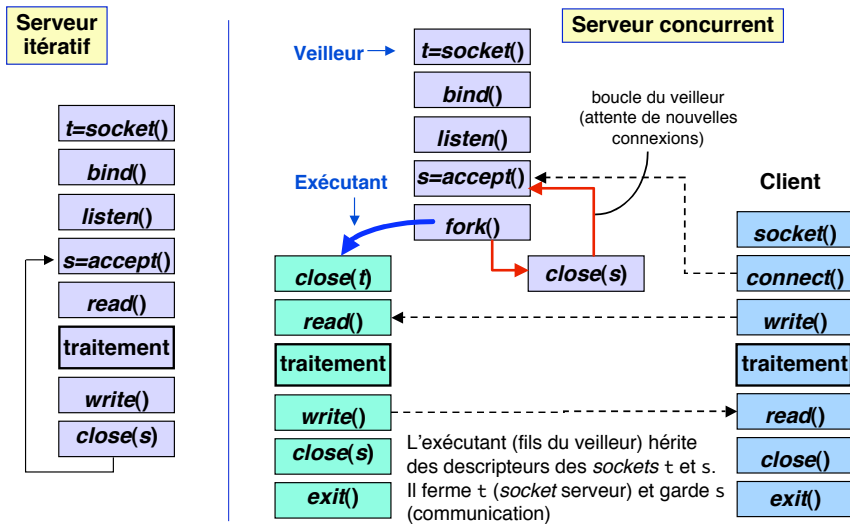
## Client-serveur en mode concurrent (1)

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`), 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);
    }
}
```

boucle d'attente du serveur itératif

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

## 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)

## 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

connexion au serveur (host:port)

envoie le contenu de buf au serveur

la boucle se termine par détection de EOF (control-D)

reçoit dans buf, puis imprime, la réponse du serveur

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

### Mode d'emploi

sur le serveur

```
{mandelbrot}server 4321 &
```

lance le serveur sur le port 4321

sur le client

```
{kernighan}client mandelbrot.imag.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.

Client et serveur peuvent s'exécuter sur deux machines différentes, ou sur la même machine. Le serveur doit être lancé **avant** le client. Expérimentation au cours du TP n°5

## 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,
          (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)

```
...
/* 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;
}
```

Remplir la structure sockaddr\_in pour serveur

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

## Résumé de la séance 7

---

### ■ Communication par *sockets* en mode connecté

- ◆ Principe
- ◆ Primitives `socket`, `bind`, `listen`, `accept`, `connect`
- ◆ Utilisation des primitives

### ■ Une bibliothèque C pour les *sockets*

- ◆ Opérations `Open_listenfd`, `Accept`, `Open_clientfd`

### ■ Programmation client-serveur avec *sockets*

- ◆ Serveur itératif
- ◆ Serveur à processus concurrents

Lire le Document Technique n° 3 (placard)