UNIVERSITÉ DE FRANCHE-COMTÉ

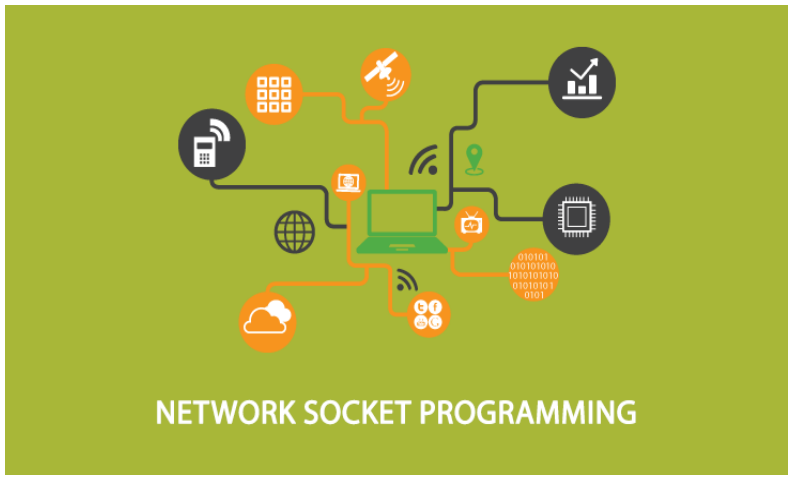
UFR SCIENCES ET TECHNIQUES BESANÇON

Programmation réseau - sockets

Définition - Types - Actions pour la communication - API

V. Felea / vafelea *at* femto-st *dot* fr / bureau 411C

Licence 3 Informatique - Dépt. Informatique / Master 1 ÉLISE - Dépt. Électronique
Faculté des Sciences et Technologies – Université Marie et Louis Pasteur



Contenu

Sockets

- Objectifs
- Définition
- Types : mode connecté / mode non connecté

Actions pour la communication

API Généralités : adresse de socket, boutisme

API Sockets

- Fonctions de communication communes TCP/UDP
- Fonctions de communication TCP
- Fonctions de communication UDP

Légende prototype fonctions : paramètres en entrée I, paramètres en sortie : O, paramètre en entrée/sortie : IO)

Service pour TCP et UDP : les sockets

besoin d'une interface de programmation d'applications réseau (API) pour Unix BSD

Objectifs

- fournir des moyens d'intercommunication entre processus (échanges locaux ou réseau)
- cacher les détails d'implémentation de la couche transport
- gérer les différences entre protocoles de transport hétérogènes sur une même interface
- simplifier la programmation

Socket

point de communication par lequel un processus peut émettre et recevoir des informations

Caractéristiques

- type (pour quel protocole de transport) : mode connecté (pour TCP), mode non connecté (pour UDP)
- adresse socket : adresse IP hôte + numéro de port
- ensemble de primitives (pour l'accès aux fonctions de transport)
- données encapsulées (un descriptif, des files d'attente de messages)

Communications orientées flux d'octets et bufferisées (1)

- la donnée applicative est divisée en segments (par le protocole de transport)
- chaque segment est envoyé individuellement (chemins potentiellement différents définis par le protocole de routage)

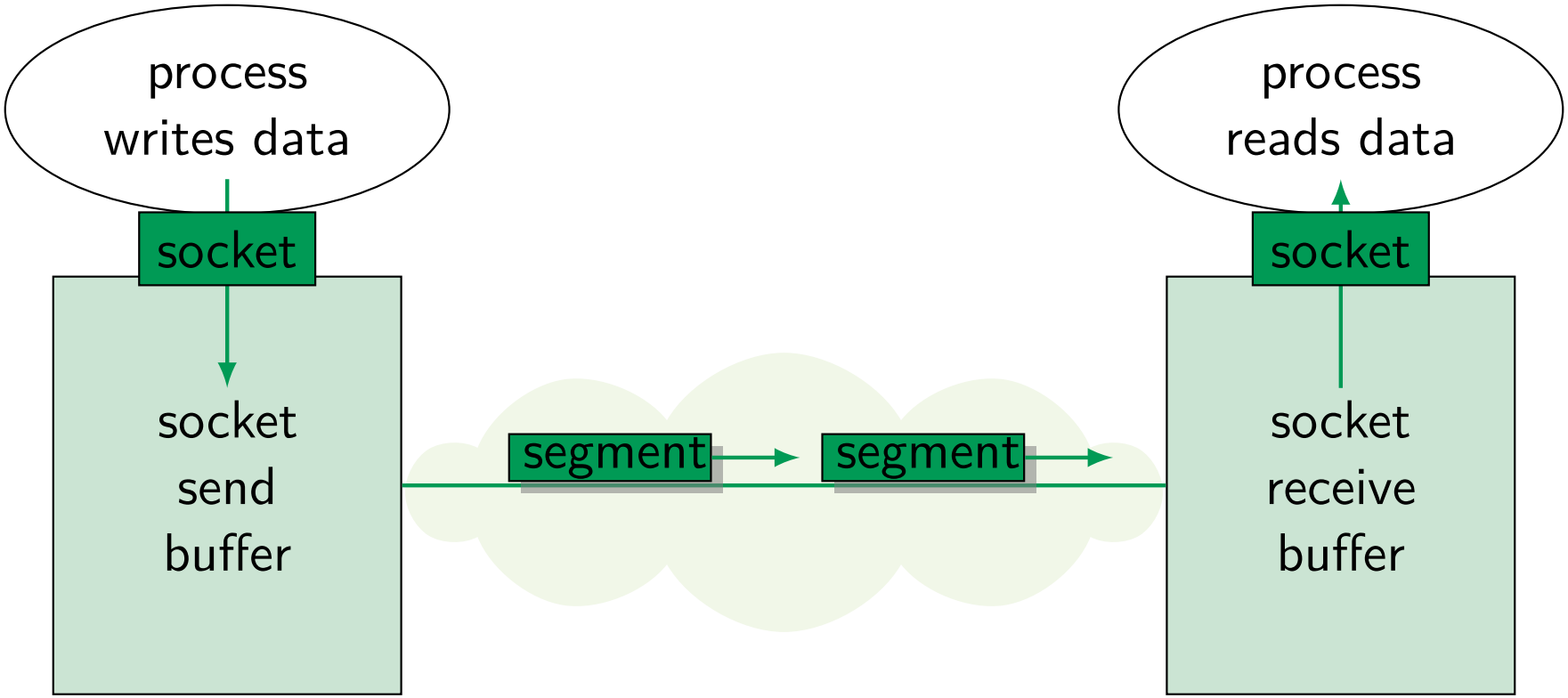
communications orientées flux d'octets

pas de typage des informations échangées (tout est octet)

communications bufferisées

les segments sont stockés dans des buffers internes du système (avant l'envoi/lors de la réception) après/avant que l'applicatif les utilise

Communications orientées flux d'octets et bufferisées (2)



Sockets en mode connecté - TCP

Caractéristiques TCP

- transport fiable en connexion, en mode bidirectionnel et point à point
- client/serveur

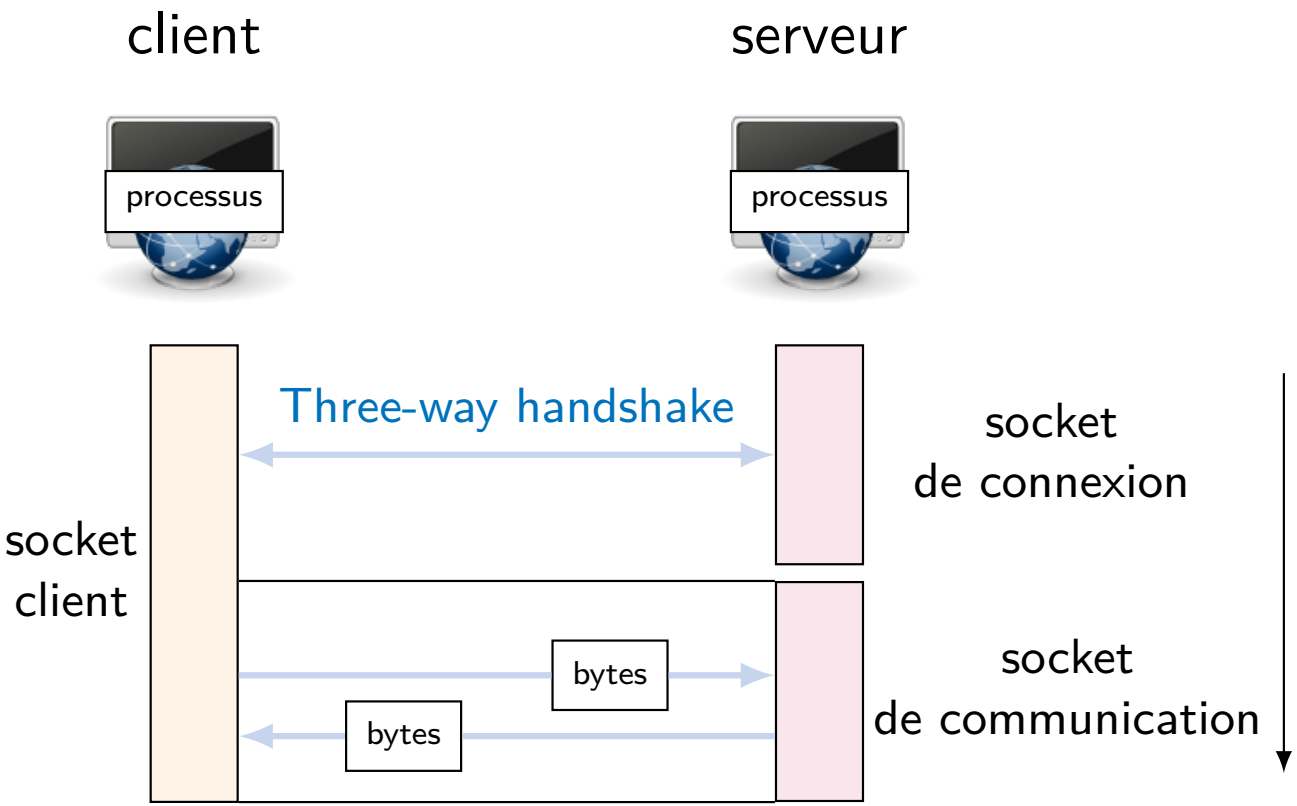
Sockets

- une socket de connexion : utilisable par plusieurs clients simultanément pour demander une connexion
- connexion : couple d'adresses de socket de communication pour deux extrémités
- socket de communication : socket à travers laquelle les données sont échangées

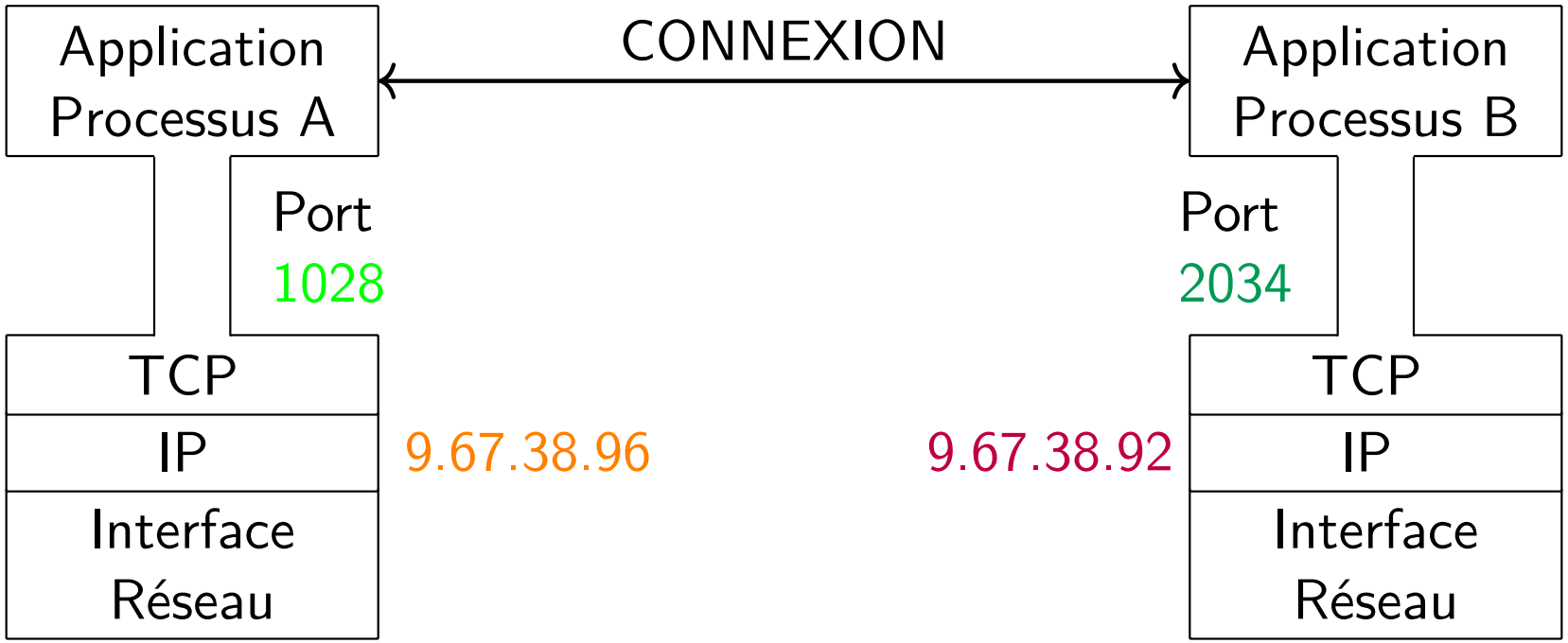
Échange de données : orienté flux d'octets et bufferisé

Les "frontières" des segments ne sont pas préservées

Socket de connexion et socket de communication

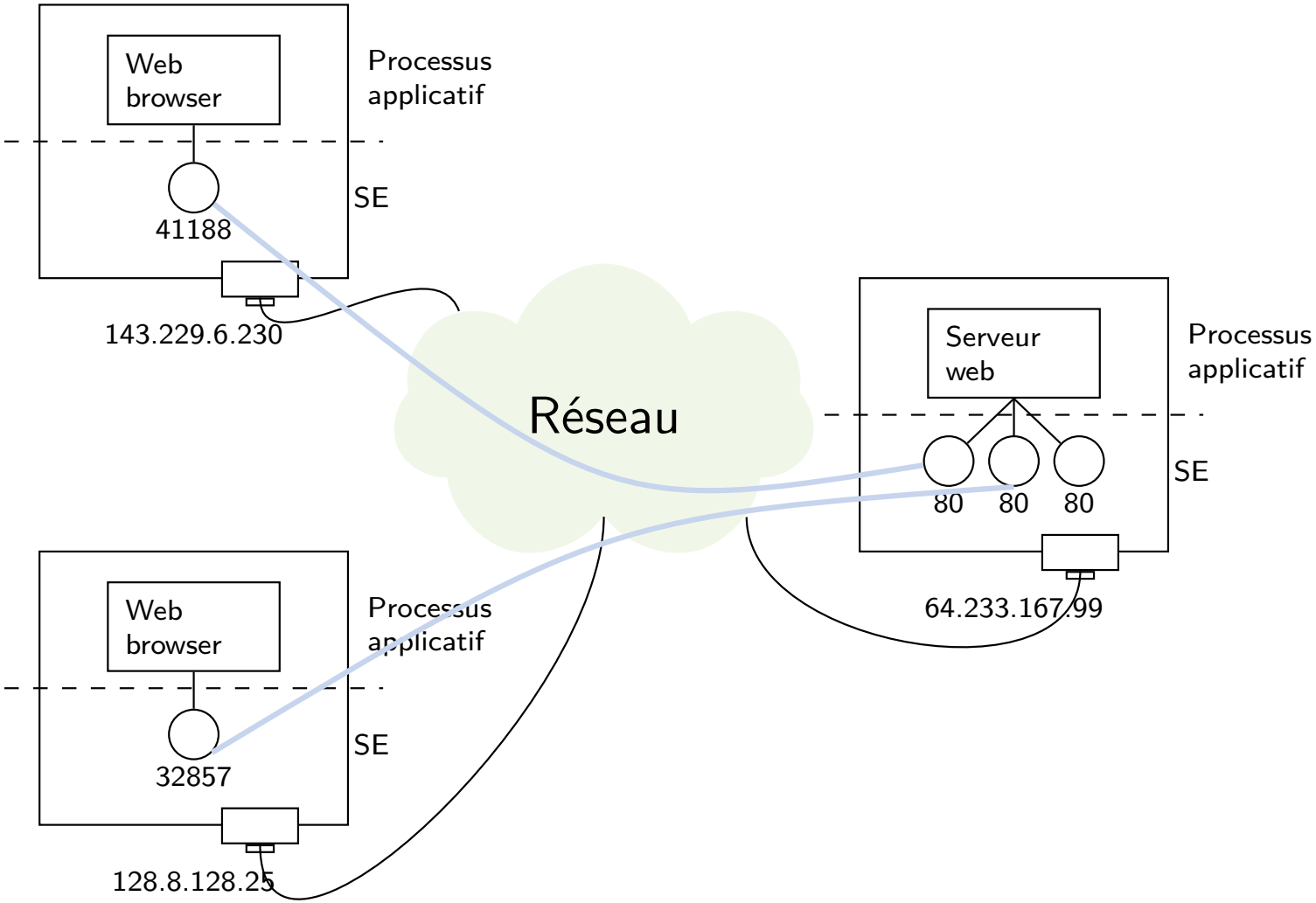


Extrémités d'une connexion



Adresse socket de A : [9.67.38.96,1028]
Adresse socket de B : [9.67.38.92,2034]

Extrémités d'une connexion - plusieurs clients



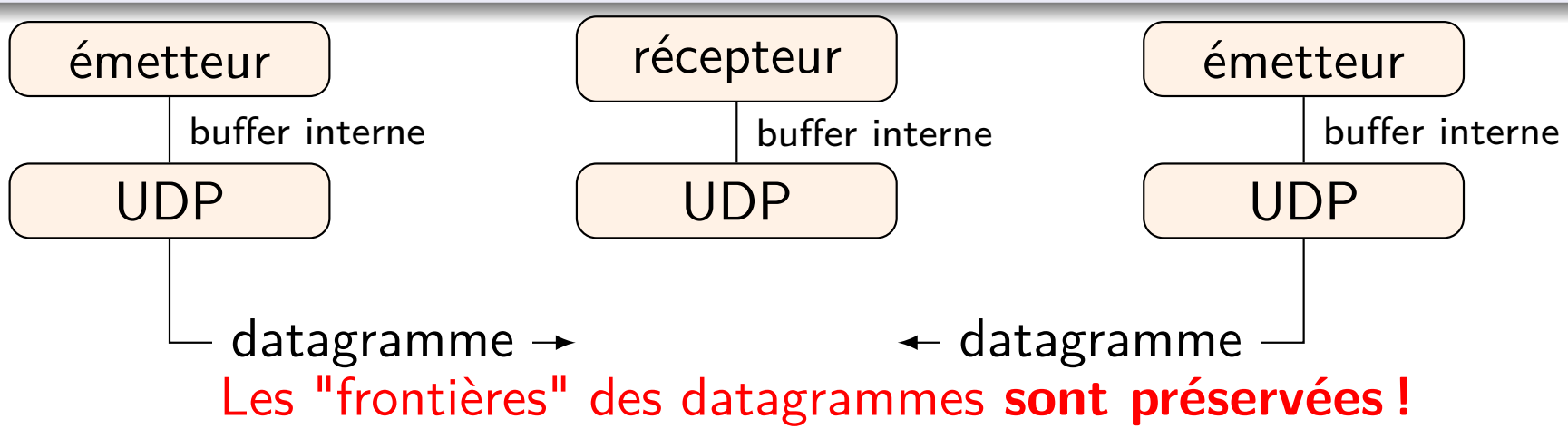
Sockets en mode non connecté - UDP

Caractéristiques UDP

- transport non fiable, sans connexion, en mode bidirectionnel
- point à point, mais point à multi-point aussi supporté (pour adresses IP de diffusion, multicast)

Échange de datagrammes

- communication bufferisées
- un buffer par socket UDP qui reçoit des datagrammes de plusieurs expéditeurs



Les "frontières" des datagrammes **sont préservées !**

Primitives C de l'interface socket

Fonctionnement TCP

- serveur :
  socket
  bind
  `listen`
  `accept`
  `send`/`recv`
  `shutdown`, **close**
- client :
  socket
  `connect`
  `send`/`recv`
  `shutdown`, **close**

Fonctionnement UDP

- émetteur/récepteur :
  socket
  bind
  `sendto`/`recvfrom`
  close

Structures et fonctions générales - adresse de socket

Adresse IPv4 de socket (netinet/in.h)

```
struct sockaddr_in {
  sa_family_t sin_family;   /* AF_UNIX/AF_INET */
  in_port_t   sin_port;     /* 16 bits, big-endian */
  struct in_addr sin_addr; /* adresse IP 32 bits, big-endian */
  char        sin_zero[8]; /* non utilisé  */
};
struct in_addr {      /* adresse IP 32 bits */
  in_addr_t s_addr;   /* bid-endian         */
};
```

Adresse générique de socket (dans les prototypes des fonctions)

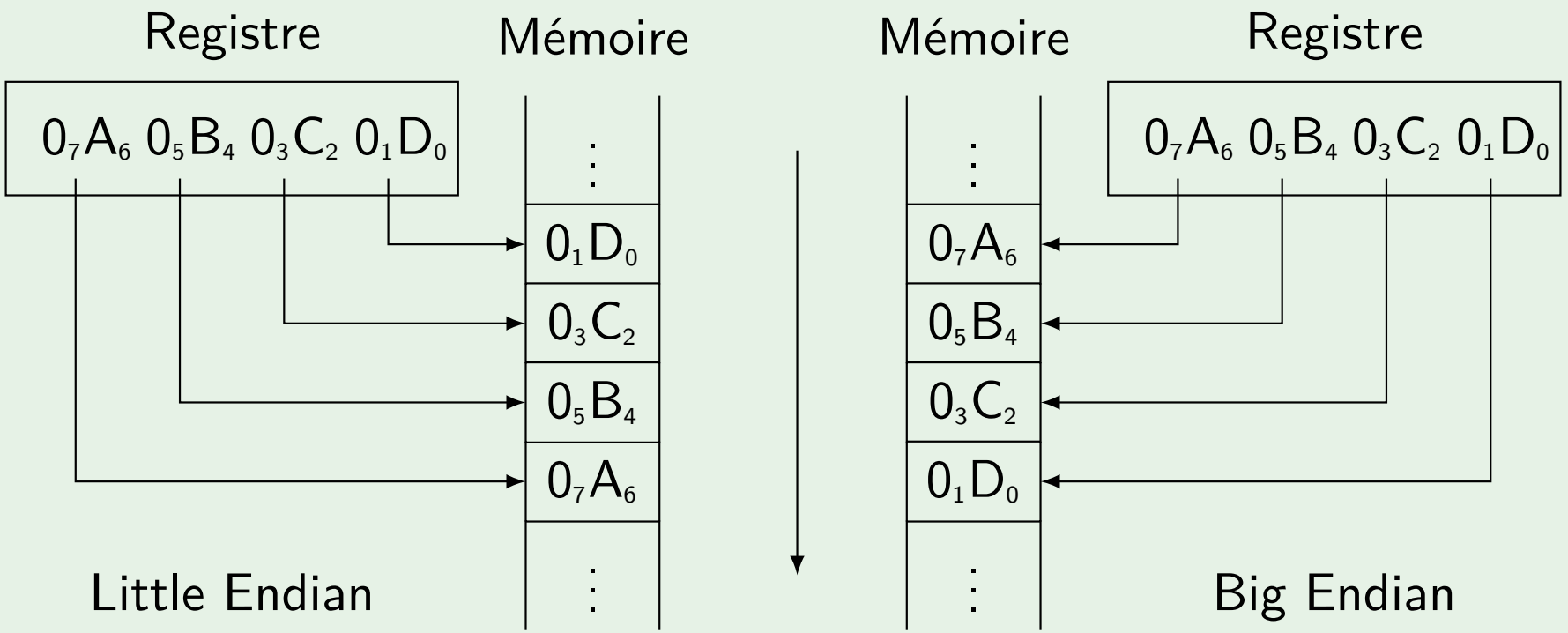
```
struct sockaddr {
  sa_family_t sa_family; /* famille d'adresse AF_xxx */
  char sa_data[14];      /* données de l'adresse     */
};
```

Boutisme

ordre des octets en mémoire

Types

- petit boutiste (little endian) : octet le moins significatif associé à l'adresse la plus basse (architectures Intel)
- gros boutiste (big endian) : octet le moins significatif associé à l'adresse la plus haute (architectures Motorola)



Ordre des octets dans les applications réseau

Données dans les protocoles

- adresse IP : big-endian
- numéro de port : big-endian

Données échangées par les applications (send/recv)

- big-endian ou little-endian (selon l'architecture matérielle)
- conversion inutile entre architectures identiques
- conversion nécessaire entre architectures différentes

Ordres prédéfinis

- ordre couche réseau : big-endian
- ordre couche applicative (hôte) : selon l'architecture (big ou little endian)

Fonctions de conversion

hôte → réseau (**h**ost **to** **n**etwork)

- `htons` : pour type short
- `htonl` : pour type long

réseau → hôte (**n**etwork **to** **h**ost)

- `ntohs` : pour type short
- `ntohl` : pour type long

Fonctions communes : **socket**

```
int socket(int family, int type, int protocol);
```

sys/types.h, sys/socket.h

- `family` (I) : spécifie la famille des protocoles (AF_INET - protocole IPv4 // AF_INET6 - protocole IPv6 // AF_UNIX - communication locale)
- `type` (I) : la sémantique de communication
  - `SOCK_STREAM` : sockets pour TCP
  - `SOCK_DGRAM` : socket pour UDP
  - `SOCK_RAW` : sockets de base
- `protocol` (I) : (généralement) 0

- créer une socket
- valeur renvoyée : le descripteur de socket, -1 en cas d'erreur

Fonctions communes : **bind**

```
int bind (int sockfd, const struct sockaddr* addr,
          socklen_t addrlen);
```

sys/socket.h

- `sockfd` (I) : descripteur de socket
- `addr` (I) : pointeur vers une adresse de socket, dont les champs :
  - `sin_family` : `AF_INET`
  - `sin_addr.s_addr` : `INADDR_ANY`
  - `sin_port` : numéro de port (**big endian**)
  - `sin_zero` : ràz (fonction `memset`)
- `addrlen` (I) : taille d'une adresse de socket
  `sizeof (struct sockaddr_in)`

- associer une adresse à une socket
- valeur renvoyée : 0 si succès, -1 en cas d'erreur

Fonctions communes : **close**

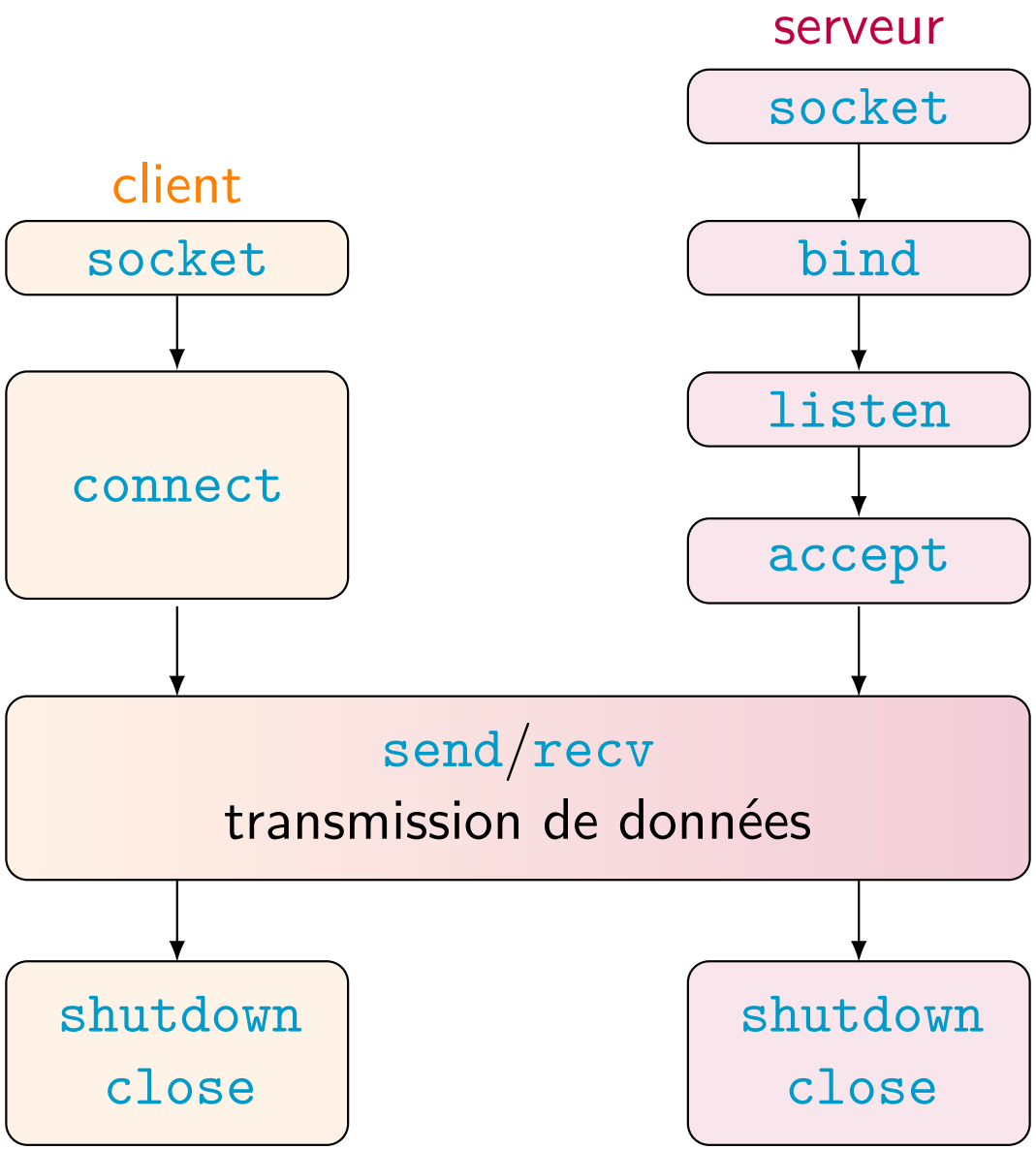
```
int close(int filedes);
```

unistd.h

- `filedes` (I) : descripteur de fichier correspondant à la socket

- fermer une socket comme descripteur de fichier (libérer le descripteur)
- valeur renvoyée : 0 si succès, -1 en cas d'erreur

Communication TCP : actions client/serveur



Client : fonction **connect** (1)

```
int connect(int sockfd, const struct sockaddr* addr,
            socklen_t addrlen);
```

sys/types.h, sys/socket.h

- `sockfd` (I) : descripteur de socket
- `addr` (I) : pointeur sur la structure qui contient l'adresse IP de la machine serveur - obtenue par **inet_aton** (IPv4) ou **getaddrinfo** (IPv4/IPv6) - et le numéro de port (big-endian)
- `addrlen` (I) : longueur de la structure pointée par `addr`

- connecter une socket à une adresse donnée (de serveur) - initier le *three-way handshake*
- valeur renvoyée : 0 si succès, -1 en cas d'erreur

Obtenir une adresse de socket (1)

```
int inet_aton (const char* cp, struct in_addr* inp);
```

netinet/in.h, arpa/inet.h

- cp (I) : une chaîne de caractères représentant une adresse IP (**v4**) en **notation décimale à point**
- `inp` (O) : adresse équivalente de type `in_addr_t` (big endian)

- convertir une adresse en notation décimale à point donnée par une chaîne de caractères en adresse de type `in_addr_t` (big-endian)
- valeur renvoyée : valeur non nulle si succès, 0 en cas d'erreur

conversion inverse : `inet_ntoa`
conversion texte vers binaire (IPv4 et IPv6) : `inet_pton`
conversion binaire vers texte (IPv4 et IPv6) : `inet_ntop`

Connexion du client

Exemple client : **connect** avec inet_aton

```
addrServ.sin_family = AF_INET;
int err = inet_aton(servIP, &addrServ.sin_addr);
/* vérifier la conversion (err)
   servIP : IPv4, en notation décimale à point */
addrServ.sin_port = htons(servPort);
bzero(addrServ.sin_zero, 8);

int err = connect(sock,
                  (struct sockaddr*)&addrServ,
                  sizeof(struct sockaddr));
```

Obtenir une adresse de socket (2)

```
int getaddrinfo (const char* node, const char* service,
                 const struct addrinfo* hints,
                 struct addrinfo** res);
```

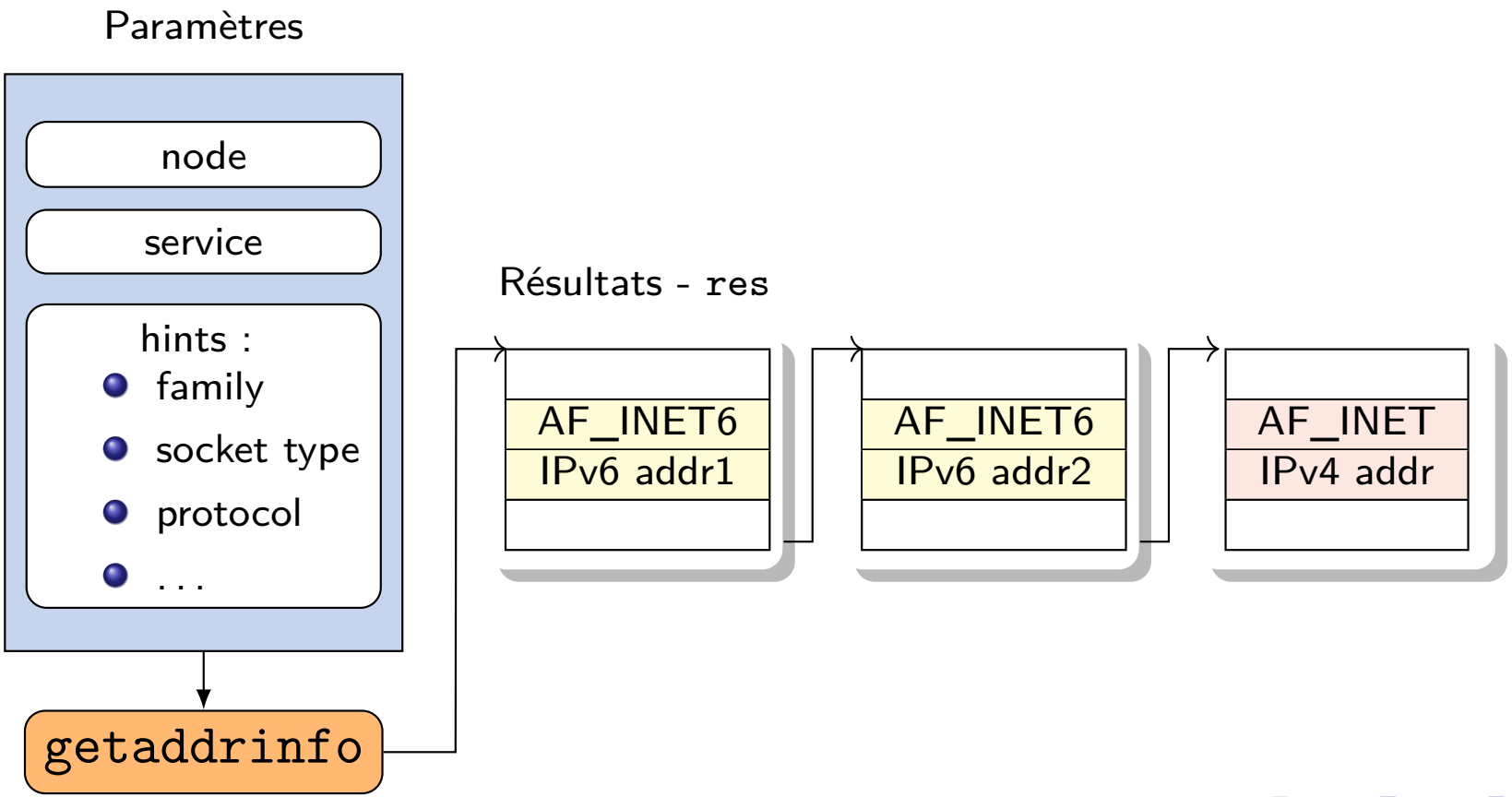
sys/types.h, sys/socket.h, netdb.h

- `node` (I) : nom d'hôte ou adresse IP (v4 ou v6)
- `service` (I) : numéro de port ou nom de service (/etc/services)
- `hints` (I) (`ai_family`, `ai_socktype`, `ai_flags`, `ai_protocol`) : (`AF_INET`/`AF_INET6`, `SOCK_STREAM`, 0, 0) + 0 pour les autres
- `res` (O) : une ou plusieurs structures `addrinfo`, chacune contenant une @IP utilisable dans les fonctions `bind`/`connect`

- convertir un nom d'hôte/adresse IP en adresse de socket
- valeur renvoyée : 0 si succès, valeur non nulle en cas d'erreur

Structure addrinfo

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



Exemple client : **connect** avec getaddrinfo

```
struct addrinfo hints, *res, *ressave;
/* préparer la structure pour obtenir les adresses IP */
bzero(&hints, sizeof(struct addrinfo));
hints.ai_family = AF_INET; hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = 0;
if((err = getaddrinfo(hostname, port, &hints, &res))!=0)
// error

ressave = res;  ok = 0;
do {
   sockfd = socket(res->ai_family, res->ai_socktype,
                   res->ai_protocol);
   if(sockfd < 0)  continue;  /* ignorer cette @ip */
   if(connect(sockfd, res->ai_addr, res->ai_addrlen)==0) {
      /* success*/  ok=1;
   }
   else  // error
      close(sockfd);/* ignorer */
} while (!ok && (res = res->ai_next) != NULL);

freeaddrinfo(ressave);
```

- tester la création de socket et la connexion au fur et à mesure sur les adresses retournées
- dès qu'une connexion est réussie les autres adresses sont ignorées

Équivalence @ip - nom d'hôte

Nom d'hôte

- chaîne de caractères
- commande système pour l'hôte courant : **hostname**

Cas général (hôte distant)

- objectif : identifier un hôte plus facilement sur internet
- historiquement : fichier *hosts* (association @IP-hôte pour machine courante et pour les autres)
- trop complexe à maintenir dans un environnement de type internet
- solution : serveur et protocole DNS (voir cours sur la couche applicative)

Serveur : fonction **listen**

```
int listen(int sockfd, int backlog);
```

sys/types.h, sys/socket.h

- `sockfd` (I) : descripteur de socket
- `backlog` (I) : longueur maximale de la file de demandes de connexions

- rendre la socket **de connexion** passive, en attente des demandes de connexions
- valeur renvoyée : 0 si succès, -1 en cas d'erreur

Serveur : fonction **accept**

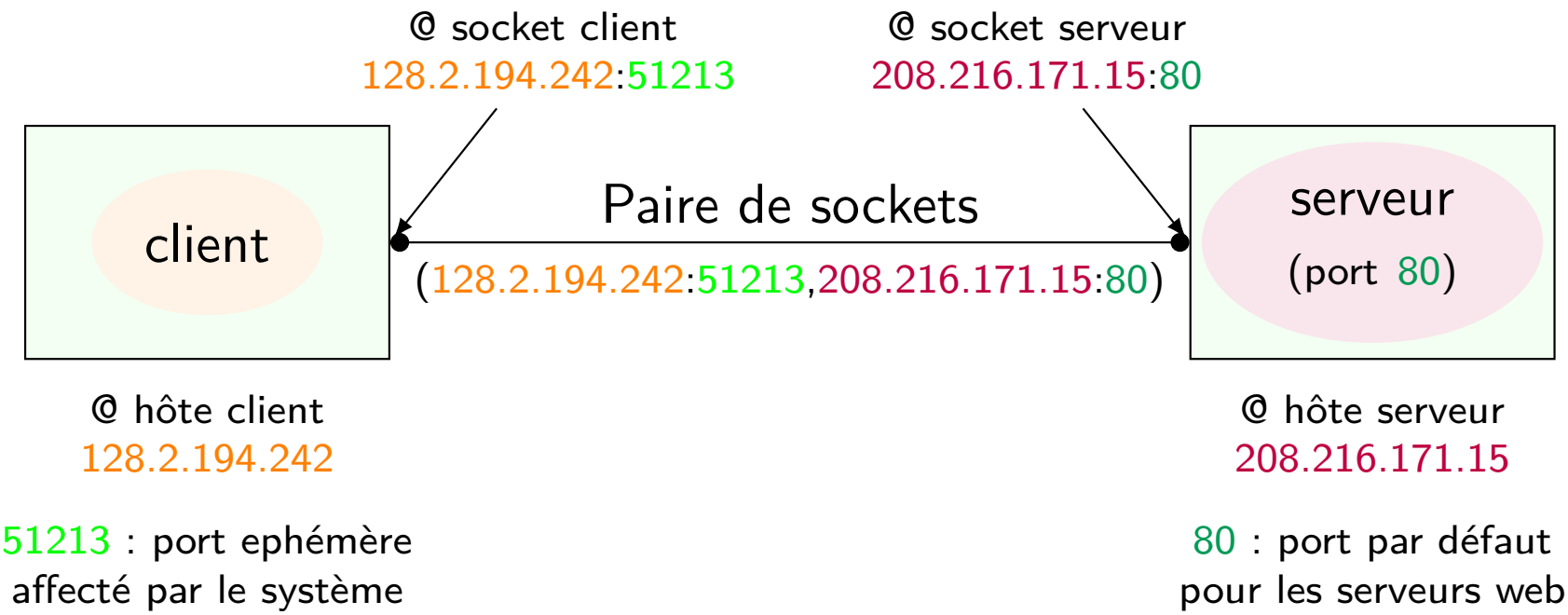
```
int accept (int sockfd, struct sockaddr* ad, socklen_t* adlen);
```

sys/types.h, sys/socket.h

- `sockfd` (I) : descripteur de socket
- `ad` (O) : adresse de la socket cliente (ou NULL si cette adresse veut être ignorée)
- `adlen` (IO) : pointeur vers la longueur de la structure pointée par `ad` / NULL

- créer une nouvelle socket connectée (*socket de communication*) en acceptant une demande de connexion, pour communiquer avec le client
- valeur renvoyée : un entier non négatif représentant le descripteur de la socket créée si succès, -1 en cas d'erreur

Structure d'une connexion



Sur le serveur, toutes le connexions établies avec les clients ont un même port associé, identique au port de la socket de connexion.

Client/Serveur : fonction **send**

```
ssize_t send (int sockfd, const void* buf,
              size_t len, int flags);
```

sys/types.h, sys/socket.h

- `sockfd` (I) : descripteur de socket
- `buf` (I) : adresse de l'espace mémoire à partir duquel l'envoi est réalisé
- `len` (I) : nombre d'octets à envoyer
- `flags` (I) : options (0)

- envoyer de données
- valeur renvoyée : nombre d'octets réellement envoyés si succès, ou -1 en cas d'erreur

Client/Serveur : fonction **recv**

```
ssize_t recv (int sockfd, void* buf,
              size_t len, int flags);
```

sys/types.h, sys/socket.h

- `sockfd` (I) : descripteur de socket
- `buf` (O) : adresse de l'espace mémoire à partir duquel les données reçues sont stockées
- `len` (I) : nombre maximum d'octets à recevoir
- `flags` (I) : options (0)

- recevoir de données
- valeur renvoyée : le nombre d'octets réellement reçus si succès, 0 si déconnexion de l'autre extrémité, ou -1 en cas d'erreur

Client/Serveur : fonction **shutdown**

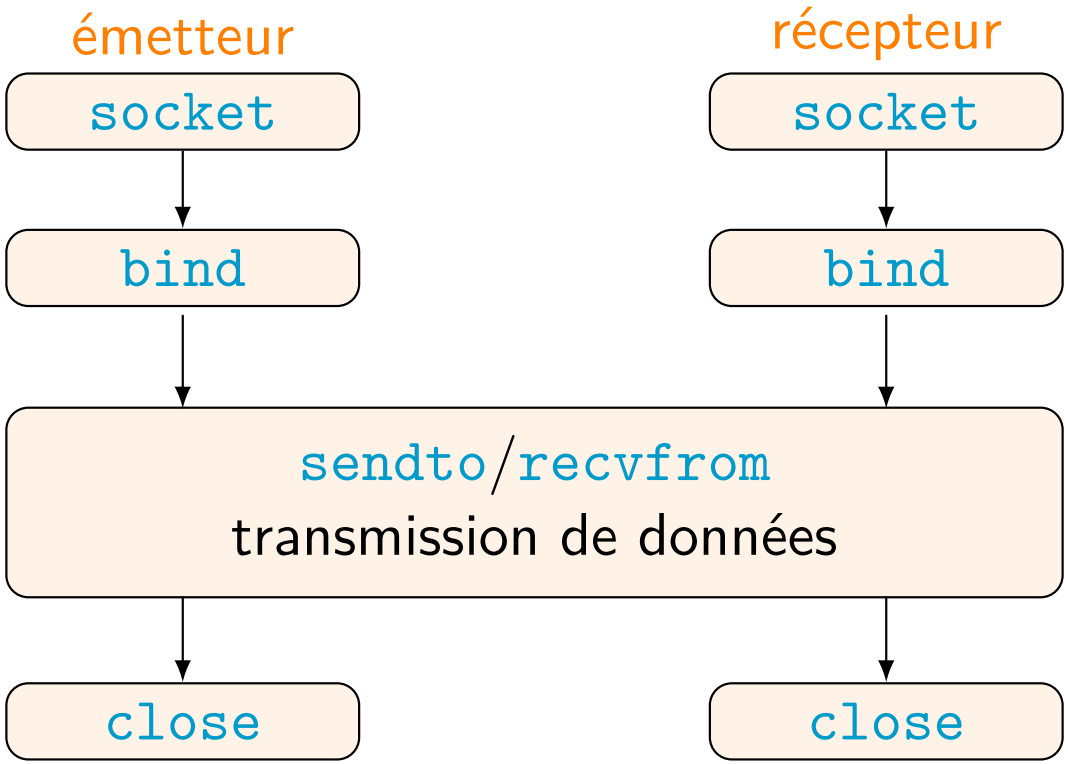
```
int shutdown (int sockfd, int how);
```

sys/socket.h

- `sockfd` (I) : descripteur de socket
- `how` (I) : manière dont la fermeture doit être réalisée
  - `SHUT_RD` : aucune autre réception
  - `SHUT_WR` : aucune autre transmission
  - `SHUT_RDWR` : aucune autre réception, ou transmission

- fermer le flux de communication associé à une socket de communication
- valeur renvoyée : 0 si succès, -1 en cas d'erreur

Communication UDP - actions émetteur/récepteur

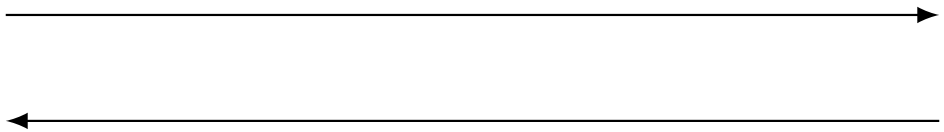


- chaque participant crée une socket locale liée à une adresse (`socket`+`bind`)
- l'envoi doit préciser le destinataire (`sendto`)
- la réception peut connaître l'expéditeur (`recvfrom`)

Exemple de transmission de données en UDP

@IP : 192.168.0.50
port : 1200

Requête
@IP source : 192.168.0.50
@IP destination : 192.168.0.150
Port source : 1200
Port destination : 21



Réponse
@IP source : 192.168.0.150
@IP destination : 192.168.0.50
Port source : 21
Port destination : 1200

@IP : 192.168.0.150
port : 21

Émetteur/Récepteur : fonction **sendto**

```
ssize_t sendto (int sockfd, const void* buf, size_t len,
                int flags, const struct sockaddr* dest_addr,
                socklen_t addrlen);
```

sys/types.h, sys/socket.h

- `sockfd`, `buf`, `len`, `flags` : identiques aux premiers paramètres de la fonction send
- `dest_addr` (I), `addrlen` (I) : concernent l'adresse du récepteur

  utiliser `inet_aton` ou `getaddrinfo` pour remplir `dest_addr`
  `addrlen = sizeof(struct sockaddr))`

- envoyer de données
- valeur renvoyée : nombre d'octets réellement envoyés, si succès, -1 en cas d'erreur

Émetteur/Récepteur : fonction **recvfrom**

```
ssize_t recvfrom (int sockfd, void* buf, size_t len,
                  int flags, struct sockaddr* src_addr,
                  socklen_t* addrlen);
```

sys/types.h, sys/socket.h

- `sockfd`, `buf`, `len`, `flags` : identiques aux premiers paramètres de la fonction `recv`
- `src_addr` (O), `addrlen` (IO) : concernent l'adresse de l'expéditeur (peuvent être NULL)

- recevoir de données et si souhaité, identifier l'expéditeur
- valeur renvoyée : nombre d'octets réellement reçus si succès, -1 en cas d'erreur

Bilan

- API sockets pour la programmation des applications en réseau
  - interface d'accès au mode de transport
- sockets mode connecté au dessus de TCP
  - une phase de connexion précède la phase d'échange de données
- sockets mode non connecté au dessus de UDP
  - l'émetteur précise le destinataire dans chaque envoi de données