# Programmation sockets

Module
Systèmes Communicants et Synchronisés
Master Informatique
1ère année

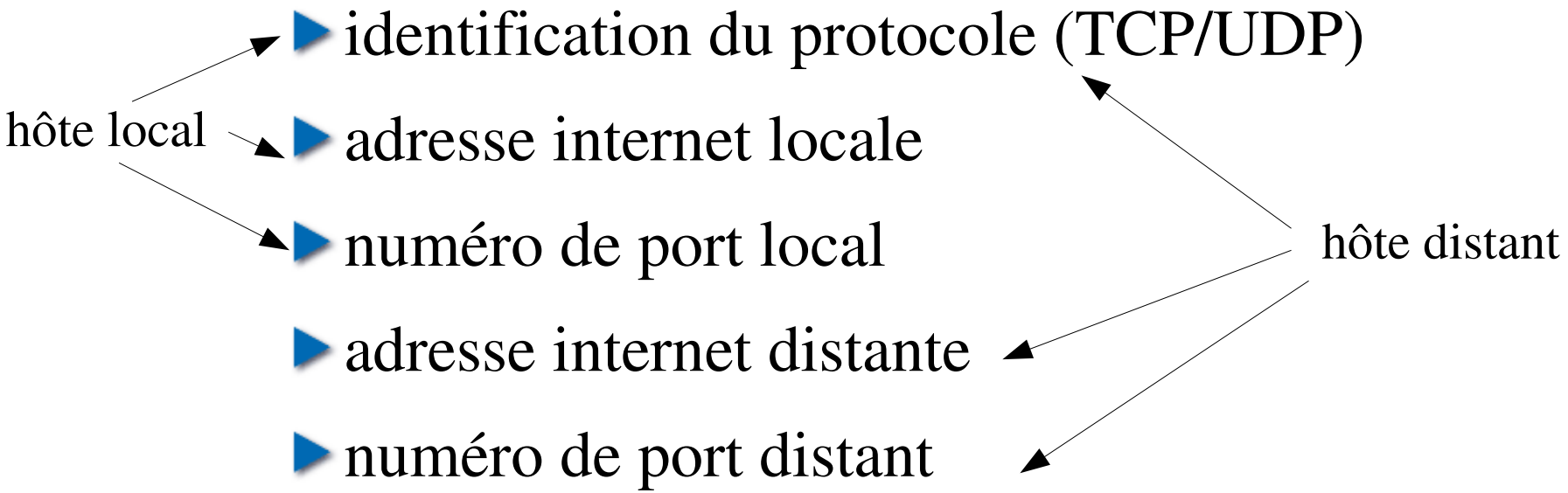
L. Philippe & V. Felea

# Programmation TCP/IP

- interaction de type client/serveur
- *client* : initie les requêtes
  - ouvre un canal de communication entre le composant client sur le hôte local et le composant serveur sur le hôte distant
  - envoie/reçoit des requêtes
  - se termine en fermant le canal de communication
- *serveur* : exécute les requêtes et renvoie les résultats
  - ouvre un canal de communication et informe de la disponibilité de recevoir des requêtes
  - traite les requêtes

# Programmation TCP/IP - association

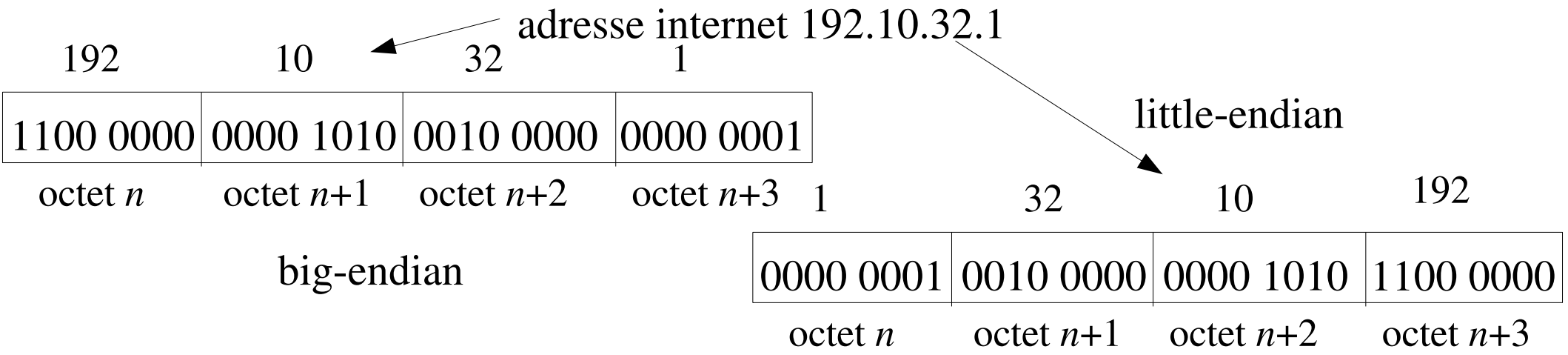
- association : la collection de 5 informations nécessaires pour l'échange d'informations entre les composants présents sur deux hôtes



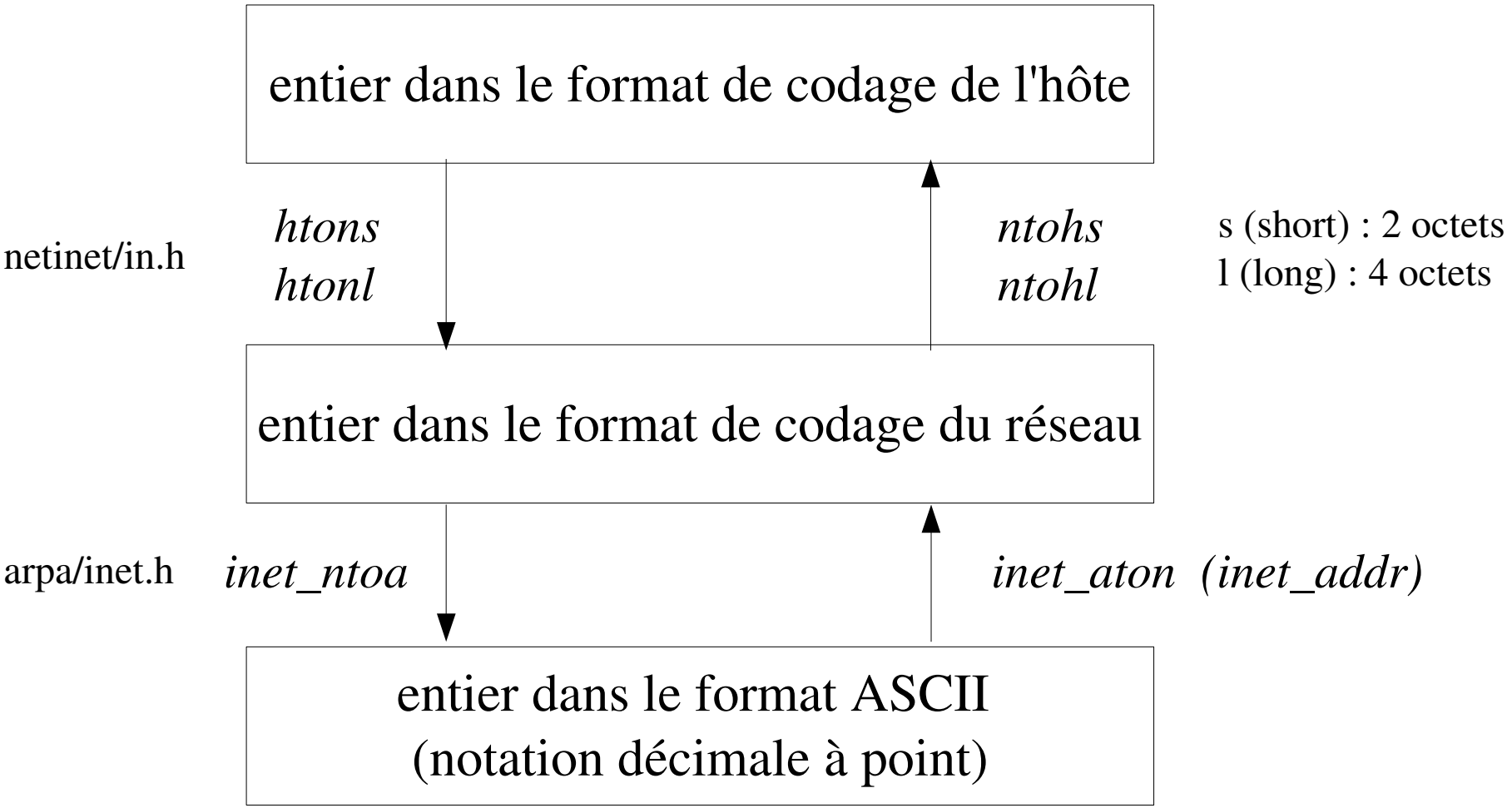

- permet la communication non ambiguë entre un processus sur le hôte local et un processus sur le hôte distant

# Codage

- l'ordre dans lequel les octets sont stockés en mémoire (adresses internet et numéros de ports)
- *big-endian* : les octets de poids fort occupent les emplacements mémoire avec des adresses plus petites (IBM mainframe, microprocesseurs Motorola)
- *little-endian* : les octets de poids faible occupent les emplacements mémoire avec des adresses plus petites (processeurs DEC VAX, microprocesseurs Intel)
- couche réseau (TCP/IP) utilise *network byte order* (big endian)



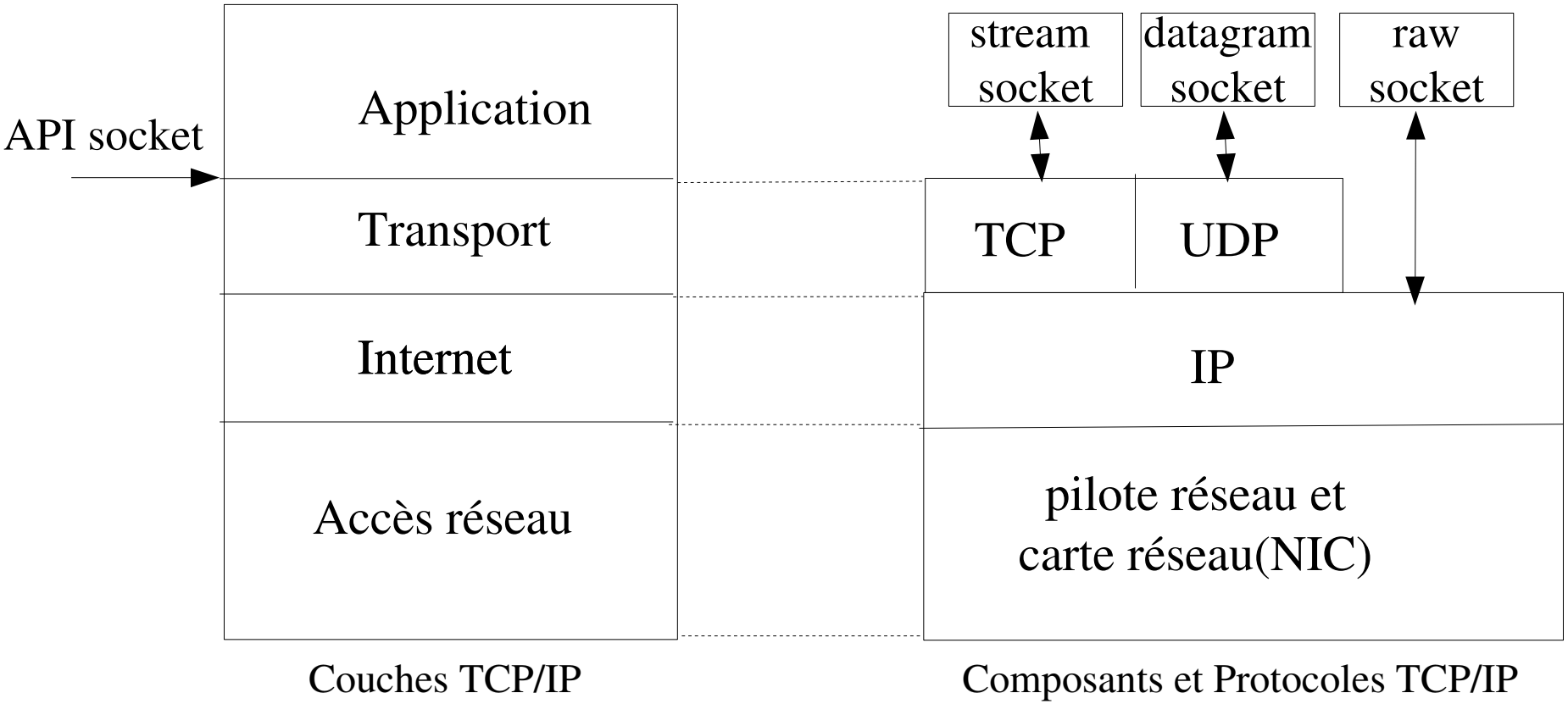
# Codage et fonctions



# Mise en œuvre du schéma de communication

- socket (bas niveau) : désigne une extrémité d'une communication
  - interface d'accès à la couche de transport (TCP/UDP)
  - permet la communication entre plusieurs processus présents sur la même machine ou sur des machines différentes
  - introduit dans Unix dans les années 80 (Berkeley Socket)
  - API (Application Programming Interface) entre les programmes d'application et la couche transport
- RPC = Remote Procedure Call (haut niveau) : appels de procédures à distance
  - des mécanismes plus évolués
    - objets répartis (ex : Java RMI, à voir en cours)
    - composants répartis (ex : EJB, Corba CCM, .Net)

# Sockets

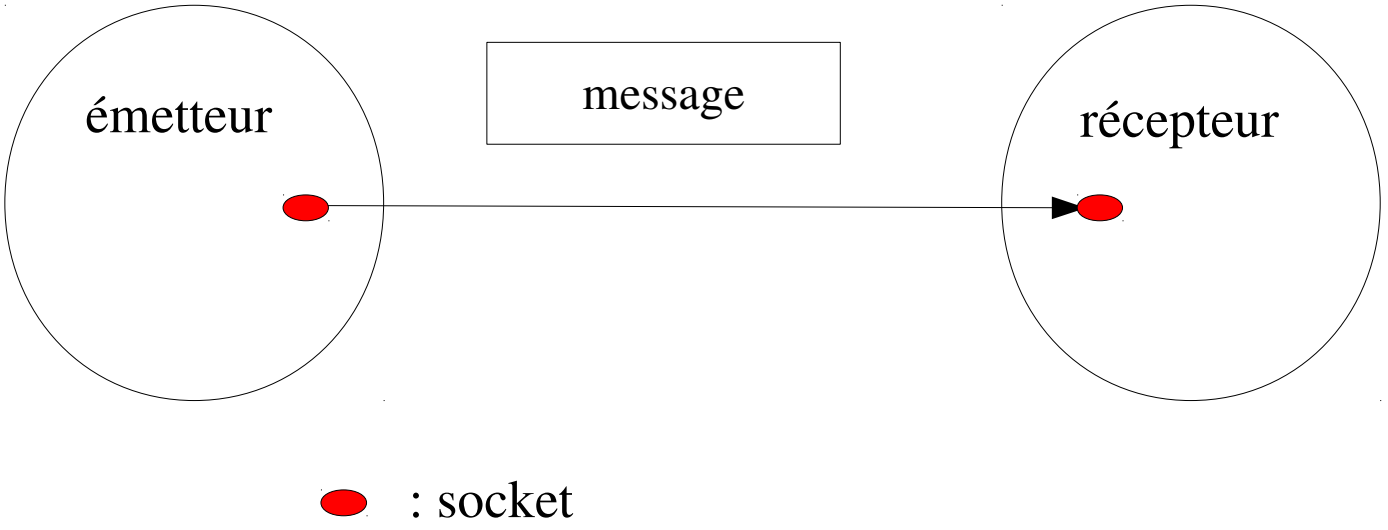

API socket
Application
Transport
Internet
Accès réseau
stream socket
datagram socket
raw socket
TCP
UDP
IP
pilote réseau et carte réseau(NIC)
Couches TCP/IP
Composants et Protocoles TCP/IP

# Programmation socket

- fonctions de communication
  - envoi : « non-bloquant », départ des données
  - réception : bloquante, arrivée des premières données
- modes de communication
  - connecté : rôles client / serveur
  - non-connecté : pas de rôles
- chacun dispose d'une socket
  - « prise » de communication, sert à l'envoi / la réception
- socket identifiée par une adresse, exemple Internet
  - adresse machine (protocole)
  - numéro de port

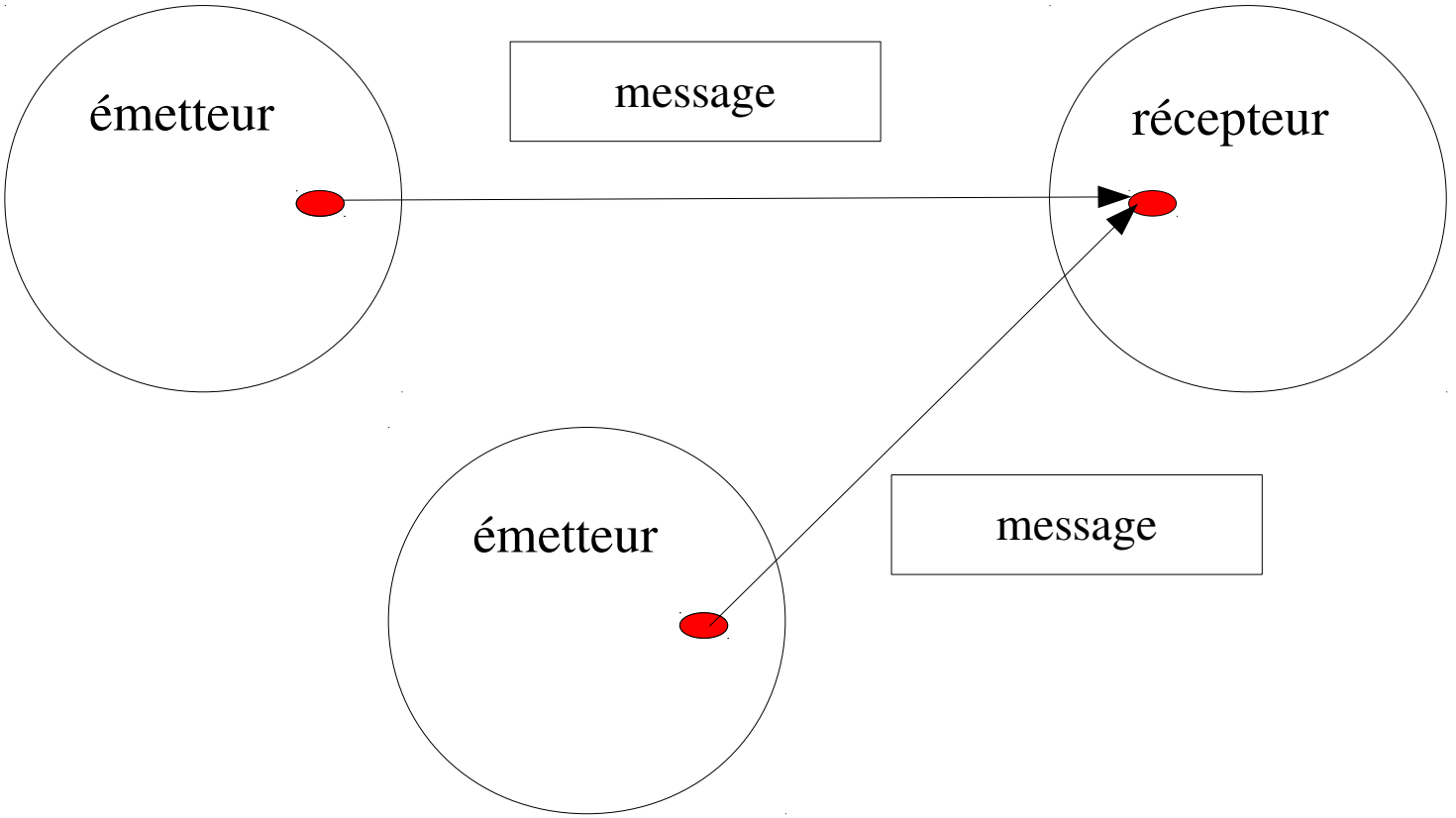
# Modèle non-connecté

- envoi non-bloquant / réception bloquante



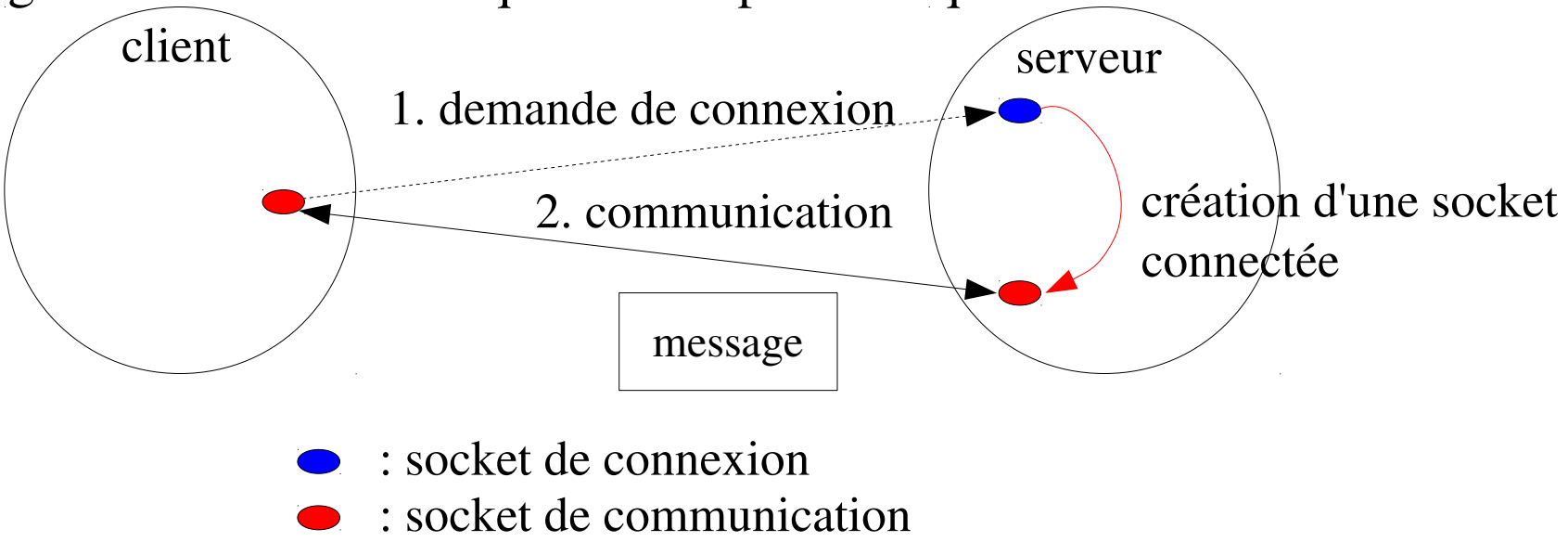

- analogie : courrier

# Modèle non-connecté



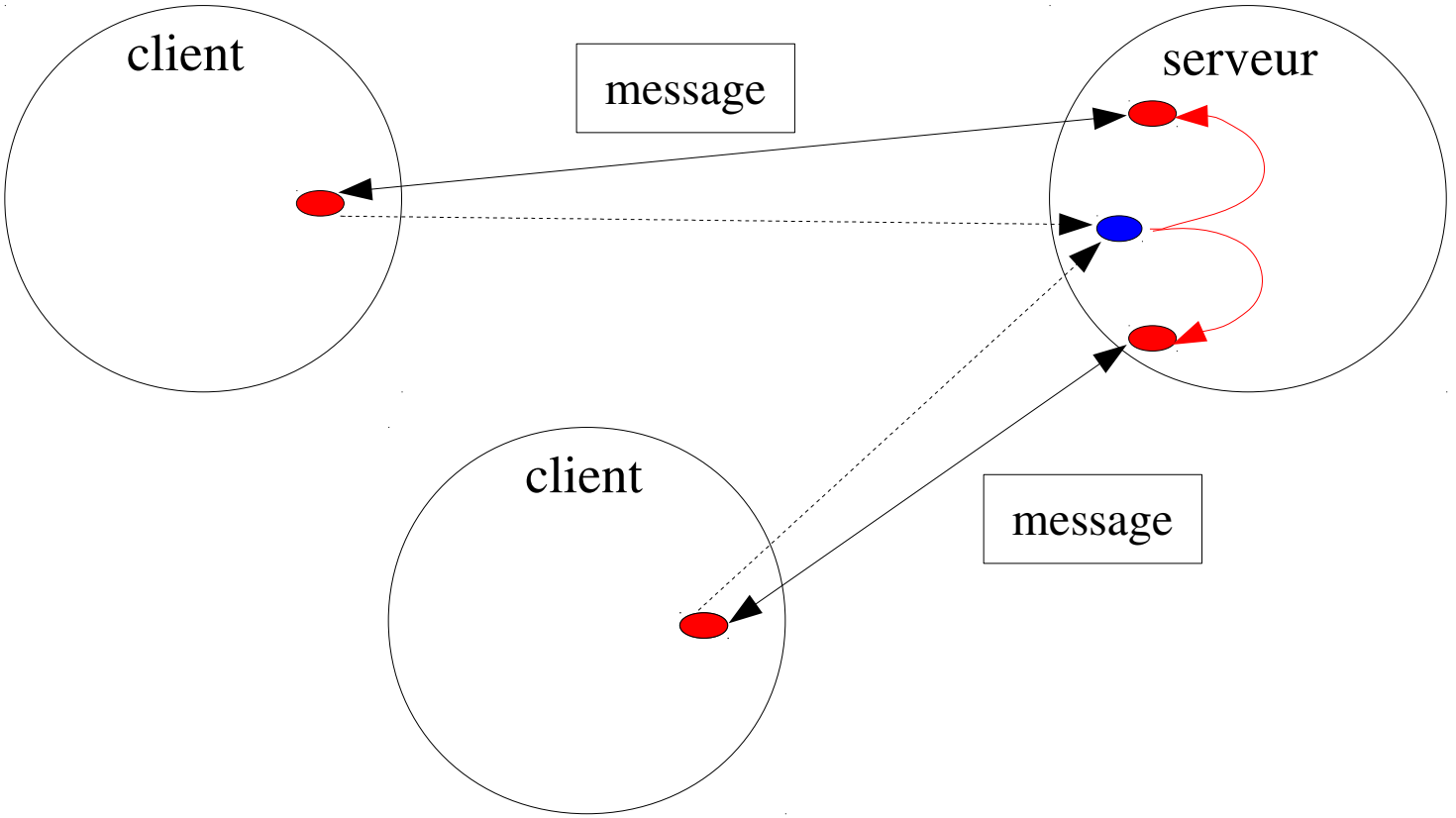
# Modèle connecté

- connecté = relation durable sans préciser de destinataire lors de l'envoi
- client = socket connectée pour l'envoi/réception
- serveur = socket de connexion + une socket par connexion
- envoi généralement non-bloquant / réception bloquante

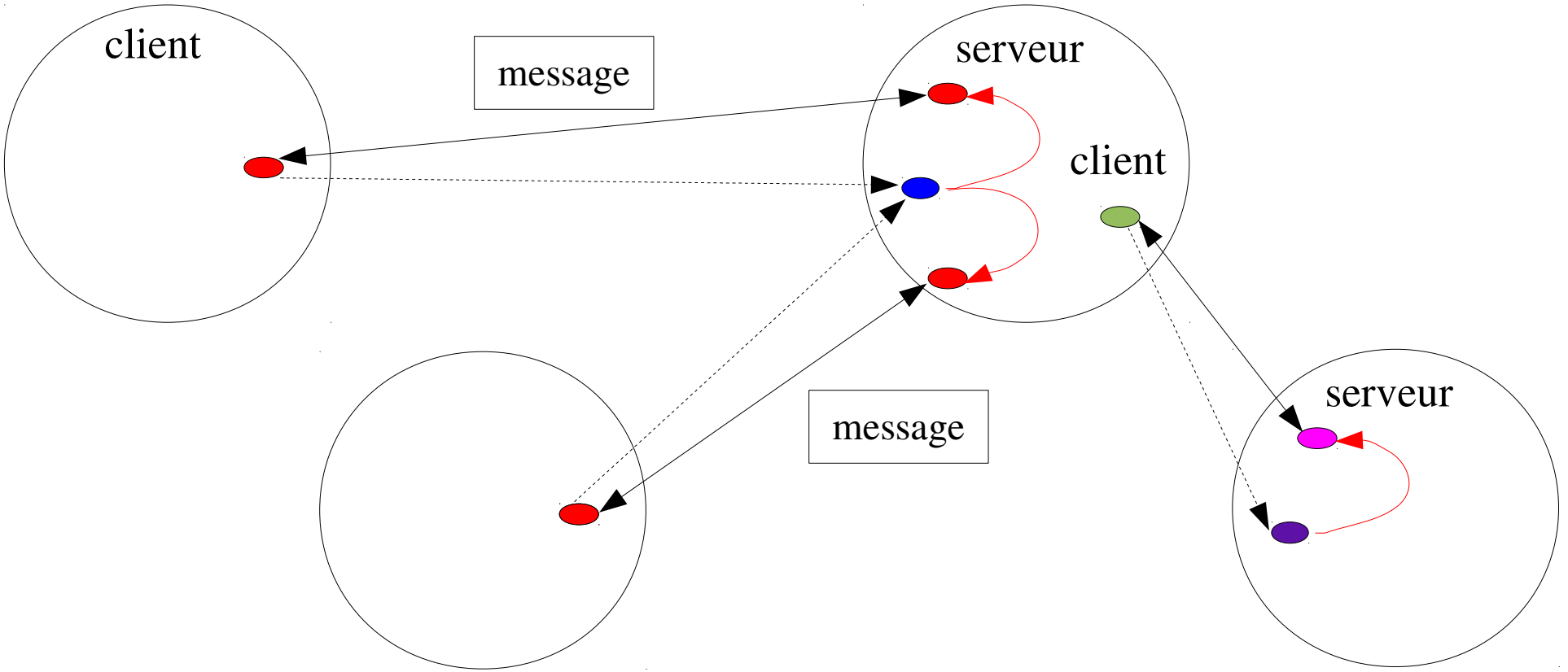



- analogie : téléphonie

# Modèle connecté



# Modèle connecté



# Généralités

- mode de communication le plus répandu (Internet)
- API socket : 1982
  - Unix BSD 4.1 / C : bas niveau
  - évolution de l'interface
- différents langages de programmation
  - C, C++, Java, .Net, Python, PHP, etc.
- différents protocoles de communication
  - TCP, IPv4, IPv6, BlueTooth, AppleTalk, X25, etc
- protocole Internet : IP
  - UDP : Unreliable Datagram Protocol = non-connecté
  - TCP : Transmission Control Protocol = connecté

# Programmation socket traditionnelle

- définition de socket : *socket*
- association d'une adresse à une socket : *bind*
- établissement d'une connexion (client) : *connect*
- mise en attente passive de connexions : *listen*
- établissement d'une connexion (serveur) : *accept*

- communications : *send*/*recv*  *sendto*/*recvfrom*

- fermeture des connexions et des sockets : *shutdown*, *close*

# Définition des sockets (1)

- *int socket(int domain, int type, int protocol)*
  - *domain* (bits/socket.h) : domaine de communication (sélectionnant une famille de protocole)
    - AF_UNIX (Unix – communication locale)
    - AF_INET (IPv4 Internet protocols)
    - AF_INET6 (IPv6)
  - *type* (bits/socket.h) : SOCK_DGRAM / SOCK_STREAM / SOCK_RAW
  - *protocol* : 0, car l'association famille-type définit généralement le protocole de transport
    - PF_INET + SOCK_STREAM => TCP = IPPROTO_TCP
    - PF_INET+ SOCK_DGRAM => UDP = IPPROTO_UDP
  - renvoie : descripteur de la socket / -1 si erreur

# Association d'une adresse à une socket

- *int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)*
  - adresse IP : sockaddr / sockaddr_in
    - obtenir l'adresse à partir du nom (DNS) (/etc/hosts)
    - *struct hostent* gethostbyname(const char *name) /*

      *int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res)*
    - recopie de l'adresse (*sin_addr.s_addr*)
  - numéro de port *(*/etc/services*)*
    - *struct servent* getservbyname(const char* name, const char* proto)*

# Structures sockaddr / sockaddr_in

- structure d'adresse générique (bits/socket.h)

```
struct sockaddr {
        unsigned short int sa_family;      /* famille */
        char   sa_data[14];                /* adresse */
};
```

- structure d'adresse AF_INET (netinet/in.h)

```
struct sockaddr_in {
        unsigned short int sin_family;
        u_short sin_port;     // encodage réseau
        struct in_addr sin_addr;
        u_char sin_zero[8];
};
```

```
struct in_addr {
        in_addr_t s_addr;
            //32-bit IPv4 address
}
```

# Structure *hostent*

- description d'une entrée pour un hôte (netdb.h)

```
struct hostent {
        char  *h_name;          /* official name of host */
        char **h_aliases;       /* alias list */
        int    h_addrtype;      /* host address type */
        int    h_length;        /* length of address */
        char **h_addr_list;     /* list of addresses */
            /* h_addr_list : tableau d'adresses de type
                                        struct in_addr*   */
};
```

# Structure *addrinfo*

- informations d'une adresse pour un fournisseur de services (netdb.h)

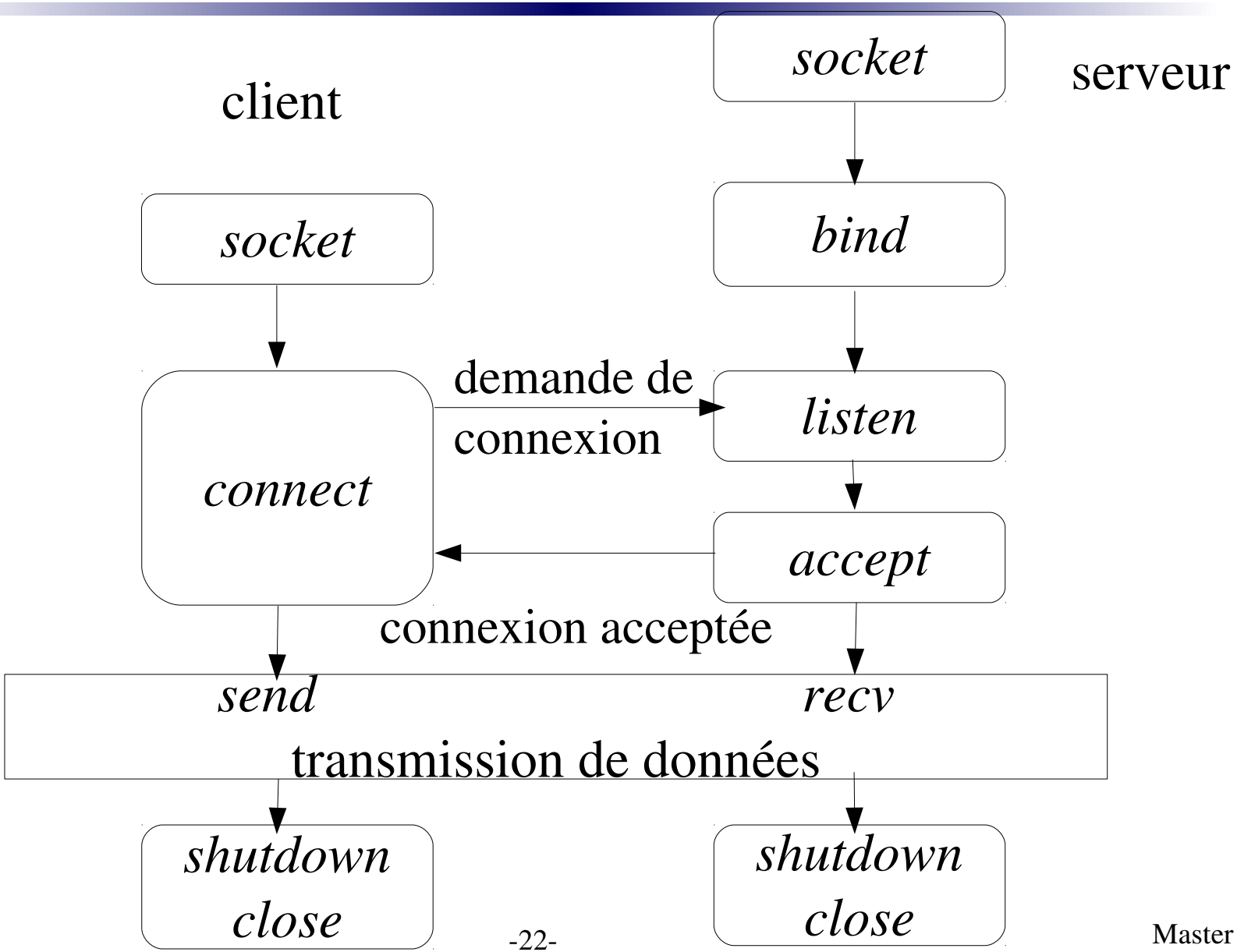
```
struct addrinfo {
  int ai_flags;              /* Input flags.  */
  int ai_family;             /* Protocol family for socket.  */
  int ai_socktype;           /* Socket type.  */
  int ai_protocol;           /* Protocol for socket.  */
  socklen_t ai_addrlen;      /* Length of socket address.  */
  struct sockaddr *ai_addr;  /* Socket address for socket.  */
  char *ai_canonname;        /* Canonical name for service location.  */
  struct addrinfo *ai_next;  /* Pointer to next in list.  */
};
```

# Structure *servent*

- structure d'une entrée pour un service (netdb.h)

```
struct servent {
        char *s_name;           /* Official service name.  */
        char **s_aliases;       /* Alias list.  */
        int s_port;             /* Port number.  */
        char *s_proto;          /* Protocol to use.  */
};
```

# Programmation socket SCS – connecté (1)



# Mise en attente de connexions (serveur)

- *int listen(int sockfd, int backlog)*
  - *sockfd* : descripteur de la socket (de type SOCK_STREAM)
  - *backlog* : définit la taille maximale de la file des connexions en attente pour la socket identifiée par le descripteur

# Programmation socket SCS – connecté (1)

- création de la connexion (serveur)
  - *int accept(int sockfd, NULL, NULL)*
  - *sockfd* : descripteur de la socket de connexion
  - *int* : descripteur de la socket connectée
- demande de connexion (client)
  - *int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)*
  - une socket créée par appel de connexion / acceptation de connexion
- autant de sockets que de connexions
  - limité par le nombre de descripteurs de fichiers
  - consultation indépendante des sockets

# Programmation socket SCS – connecté (2)

- simplification inspirée de Java
  - bibliothèque fonctionsTCP.h .c
- deux types de sockets
  - socket serveur (socket de connexion)
    - regroupe : *socket* + *bind* + *listen*
    - fonction : *int socketServeur(ushort no_port)*
    - retourne : socket de connexion (int)
    - utiliser : *int accept(int sockfd, NULL, NULL)*
  - socket client (socket de communication)
    - regroupe : *socket* + *connect*
    - fonction : *int socketClient(char* nom_mach, ushort no_port)*
    - retourne : socket communication (int)

# Programmation socket SCS – connecté (3)

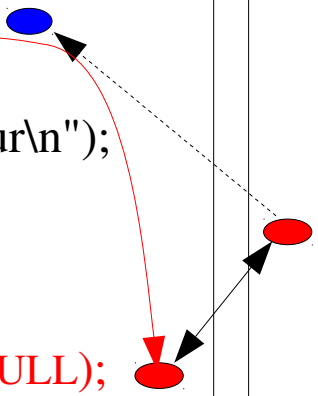
- utilisation

**Serveur**

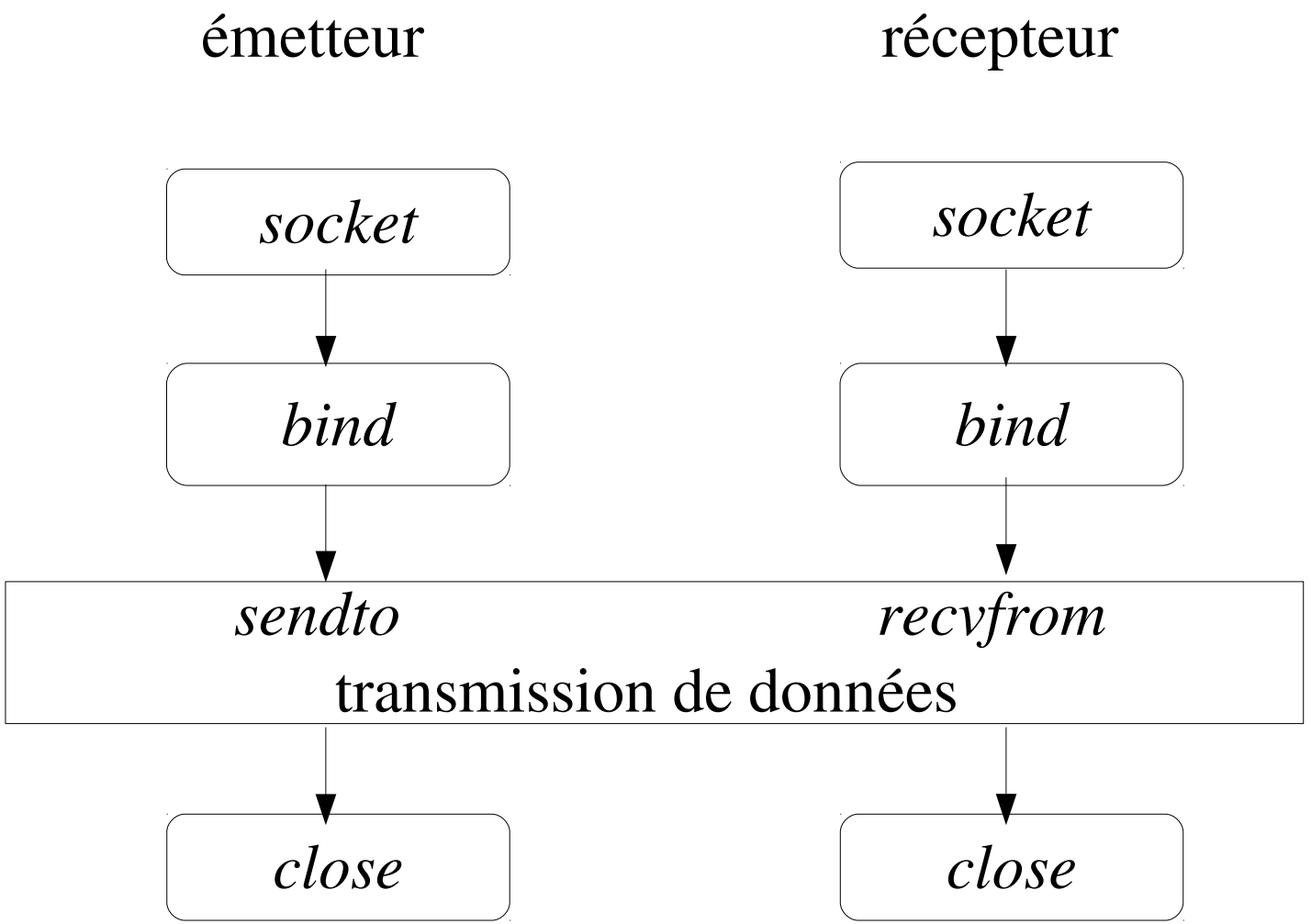
```
#include "fonctionsTCP.h"
main(int argc, char** argv) {
   int sockCon, sockTrans;  /*desc sockets*/
   int port = 6767;

   sockCon = socketServeur(port);
   if (sockCon < 0) {
      printf("serveur : erreur socketServeur\n");
      exit(2);
   }
   /*   établissement de la connexion     */
   sockTrans = accept(sockCon, NULL, NULL);
   if (sockTrans < 0) {
      printf("serveur :  erreur sur accept");
      exit(3);
   }
```

**Client**

```
#include "fonctionsTCP.h"
main(int argc, char** argv) {
  int sock,              /* desc socket locale */
      port=6767;      /* port serveur */

  char* machine ; /* l'initialiser au nom de la
                          machine serveur */

  /* création d'une socket */
  sock = socketClient(machine, port);
  if (sock  < 0) {
    printf("client : erreur socketClient\n");
    exit(2);
  }
```



# Programmation socket SCS – non connecté (1)



# Programmation socket SCS – non connecté (2)

- simplification de la création
- un seul type de sockets
  - socket de communication
    - regroupe : *socket* + *bind*
    - fonction : *int socketUDP(short no_port)*
    - retourne : socket locale (émetteur / récepteur)
  - création d'adresse pour l'envoi
    - *int adresseUDP(char* nom_mach, short no_port, struct sockaddr_in* addr)*
    - *struct sockaddr_in* initAddr(char* nom_mach, short no_port)* *// allocation mémoire pour l'adresse*
- bibliothèque : fonctionsUDP.h .c

# Programmation socket SCS – non connecté (3)

- une socket par programme
  - tous les messages sur la même socket
  - identifier la provenance des messages
- une socket par échange
  - bien gérer les attributions de numéros de ports

# Programmation socket SCS – non connecté (4)

- utilisation

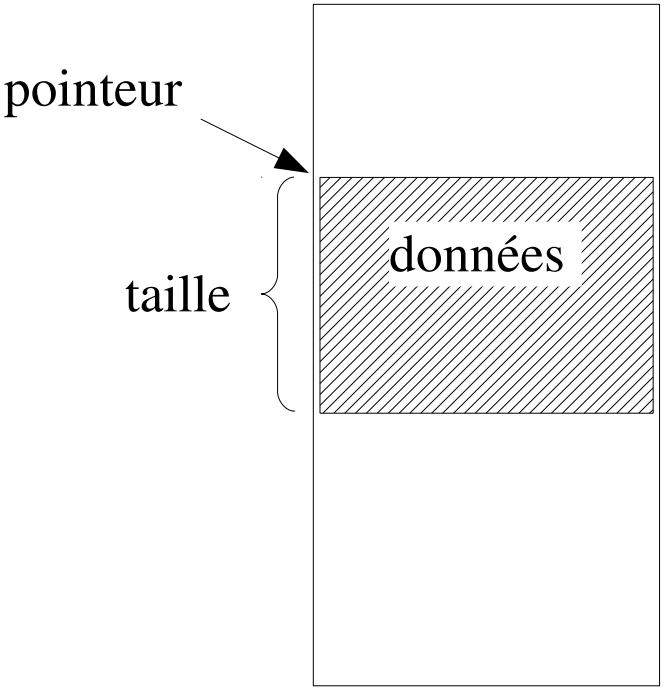
```
#include "fonctionsUDP.h"
main(int argc, char** argv) {
   int sock,  /* descripteur socket */
       port,
       err;
 char nomMach[20];   /* nom de la machine réceptrice */
 struct sockaddr_in addrDest;

 /* création de la socket */
 sock = socketUDP(6767);
 if (sock < 0) {
  printf("emetteur : erreur de socketUDP \n" );
  exit(2);
 }
  /* saisie et init de l'adresse du destinataire  */
 printf("emetteur : donner la machine dest : ");
 scanf("%s", nomMach);
 printf("emetteur : donner le port dest : ");
 scanf("%d", &port);
 err = adresseUDP(nom_mach, port, &addrDest);
 if (err  == -1) {
  printf ("emetteur : erreur adresse\n"); close(sock);
  exit(3);
 }
```

# Communication avec les sockets (1)

- envoi d'une socket à une autre
  - géré en dessous par le protocole de communication
  - échange full-duplex
  - pas de lien client/envoi et serveur/reçoit
- identification de la socket utilisée pour l'envoi
  - descripteur de socket : *int*
    - descripteur de fichier = numéro dans la table
    - local au processus
    - nombre limité (RLIMIT_NOFILE bits/resource.h)
- identification de socket de destination
  - uniquement en mode non-connecté
  - adresse *sockaddr*

# Communication avec les sockets (2)

- données envoyées
  - quelque soit le mode de connexion
  - zone de mémoire : pointeur
  - nombre d'octets (taille)
  - données brutes
    - pas de type
    - pas de frontière
  - recopiées dans le système



# Communication avec les sockets (3)

- exemples de données
  - char* message = "bonjour": message + taille (*strlen*)
  - char buffer[100] : buffer + taille (*sizeof*)
  - int value : &value + taille (*sizeof*)
  - int tabInt[20] : tabInt + taille (*sizeof*)
- calcul de la taille des données
  - à la main, connue
  - *sizeof* : type de la variable
    - *sizeof(pointeur) = taille pointeur (4 en 32 bits, 8 en 64)*
    - tableau est type de données, pas pointeur sur chaîne
  - *strlen* : uniquement chaîne (*'\0'*)
- attention : pointeur initialisé, mémoire allouée, etc

# Fonction d'envoi – mode connecté (1)

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

- *sockfd* : descripteur de la socket
- *buf* : pointeur sur les données
- *len* : nombre d'octets à envoyer
- *flags* : 0 (indication sur les propriétés de l'envoi)
- retourne
  - nombre d'octets envoyés
  - si 0 fin de connexion
  - -1 en cas d'erreur
  - toujours tester le retour
- garantie d'acheminement dans le bon ordre

# Fonction d'envoi – mode connecté (2)

- exemples

```
char chaine[100];
…
printf("client : donner la chaine : ");
scanf("%s", chaine);
…
// envoi de la chaîne
err = send(sock, chaine, strlen(chaine), 0);
if (err != strlen(chaine))  // erreur
…
```

```
int value;
…
printf("client : donner la valeur : ");
scanf("%d", &value);
…
// envoi de l'entier
err = send(sock, &value, sizeof(value), 0);
if (err != sizeof(value))  // erreur
…
```

```
int tabInt[50];
…
for (i = 0 ; i < 50 ; i++)   tabInt[i] = i;
…
// envoi du tableau
err = send(sock, tabInt, sizeof(tabInt), 0);
if (err != sizeof(tabInt))  // erreur
…
```

# Fonction d'envoi – mode non-connecté (1)

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

- *sockfd*, *buf*, *len*, *flags* et retour idem
- *dest_addr* : adresse du destinataire
  - utiliser *initAddr* ou *addresseUDP (fonctionsUDP.h)*
- *addrlen* : taille de l'adresse
  - utiliser *sizeof (dest_addr)* / *sizeof (struct sockaddr_in)*
- aucune garantie

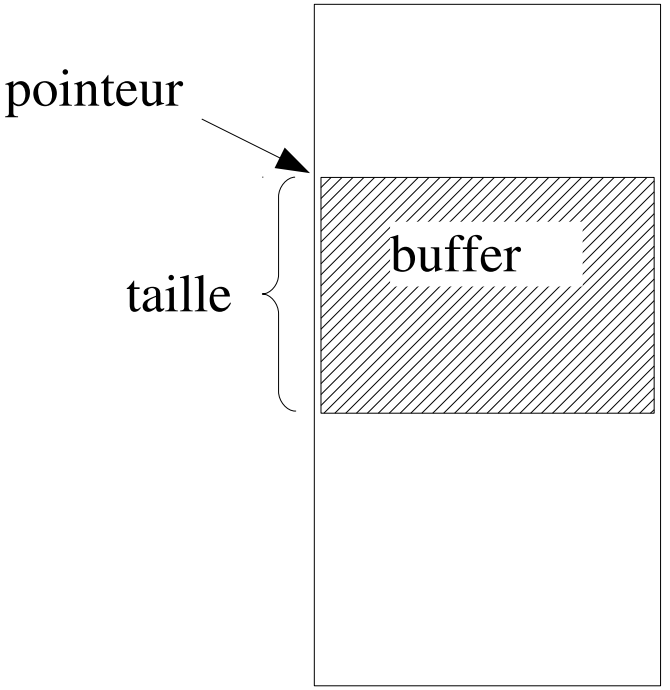
# Fonction d'envoi – mode non connecté (2)

- exemple

```
// réservation de la mémoire nécessaire au message
 char chaine[100];
 …
// initialisation du message
printf("emetteur : donner la chaine : ");  scanf("%s", chaine);
…
// initialisation de l'adresse du destinataire
struct sockaddr_in addrDest;
printf("emetteur : donner la machine dest : ");   scanf("%s", nomMachine);
printf("emetteur : donner le port dest : ");          scanf("%d", &port);
err = adresseUDP(nomMachine, port, &addrDest);
if (err  == -1) {
  printf ("emetteur : erreur adresse\n");      exit(3);
}
// envoi du message
err = sendto(sock, chaine, strlen(chaine)+1, 0,
                    (struct sockaddr*)&addrDest, sizeof(struct sockaddr_in));
if (err != strlen(chaine)+1)   // erreur
…
```

# Communication avec les sockets (4)

- réception des données (quelque soit le mode de connexion)
  - prévoir un espace de réception
  - buffer en mémoire
    - pointeur
    - nombre d'octets (taille)
  - données inconnues à l'avance
    - taille et type
  - recopiées par le système
  - ne recopie pas plus que taille
    - données restantes conservées dans la socket
    - découpage du message



# Communication avec les sockets (5)

- exemples de buffers
  - char buffer[100] : buffer + taille (*sizeof*)
  - int value : &value + taille (*sizeof*)
  - int tabInt[50] : tabInt + taille (*sizeof*)
- calcul de la taille du buffer
  - à la main, connue
  - *sizeof* : type de la variable
    - *sizeof(pointeur) = taille pointeur (4 en 32 bits, 8 en 64)*
  - *strlen* : uniquement chaîne (*'\0'*)
- attention : pointeur initialisé, mémoire allouée, etc

# Fonction de réception – mode connecté (1)

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

- *sockfd* : descripteur de socket utilisée pour recevoir
- *buf* : buffer de réception
- *len* : taille du buffer
- *flags* : propriétés de la réception
- *ssize_t* : taille du message reçu (pas de marquer de fin)
  - > 0 : taille
  - = 0 : déconnexion
  - = -1 : erreur

# Fonction de réception – mode connecté (2)

- exemples
  - données quelconques
    - chaîne
    - suite de valeurs

```
char buffer[100];  // buffer de réception
…
// réception d'une chaîne
err = recv(sockTrans, buffer,
                          sizeof(buffer), 0);
if (err < 0)  // erreur
    …
```

  - données connues

```
int valRecue;  // buffer de réception
…
// réception d'un entier
err = recv(sockTrans, &valRecue, sizeof(valRecue),0);
if (err < 0)  // erreur
    …
```

```
int tabInt[50];  // buffer de réception
…
// réception d'un tableau d'entiers
err = recv(sockTrans, tabInt,
                    sizeof(tabInt), 0);
if (err < 0)  // erreur
    …
```

# Fonction de réception – mode non connecté (1)

*ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,*

*struct sockaddr *src_addr, socklen_t *addrlen)*

- *sockfd, len, flags* : idem *recv*
- *ssize_t* : taille des données reçues / -1 si erreur
- *src_addr, addrlen : NULL*

# Fonction de réception – mode non connecté (2)

- exemples
  - données quelconques
    - chaîne
    - suite de valeurs

```
char buffer[100];  // buffer de réception
…
 // réception d'une chaîne
 err = recvfrom(sockTrans, buffer,
                sizeof(buffer), 0, NULL, NULL);
  if (err < 0)  // erreur  …
```

  - données connues

```
int valRecue;  // buffer de réception
…
// réception d'un entier
err = recvfrom(sockTrans, &valRecue,
                    sizeof(valRecue), 0, NULL, NULL);
if (err < 0)  // erreur
    …
```

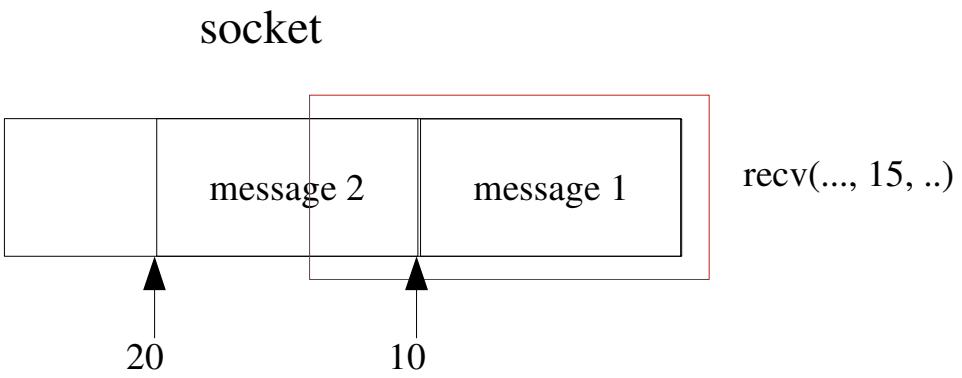
```
int tabInt[50];  // buffer de réception
 …
// réception d'un tableau d'entiers
err = recvfrom(sockTrans, tabInt,
                sizeof(tabInt), 0, NULL, NULL);
 if (err < 0)  // erreur
     …
```

# Différence échange messages – modes connecté / non connecté

- mode connecté : un seul envoi peut être consommé en plusieurs réceptions, bufferisation chez le client et chez le serveur (FIFO), pas de séparation de messages
- mode non connecté : un envoi est consommé par une seule réception (perte de données si buffer de réception pas suffisamment grand), taille maximale trame UDP : 65K

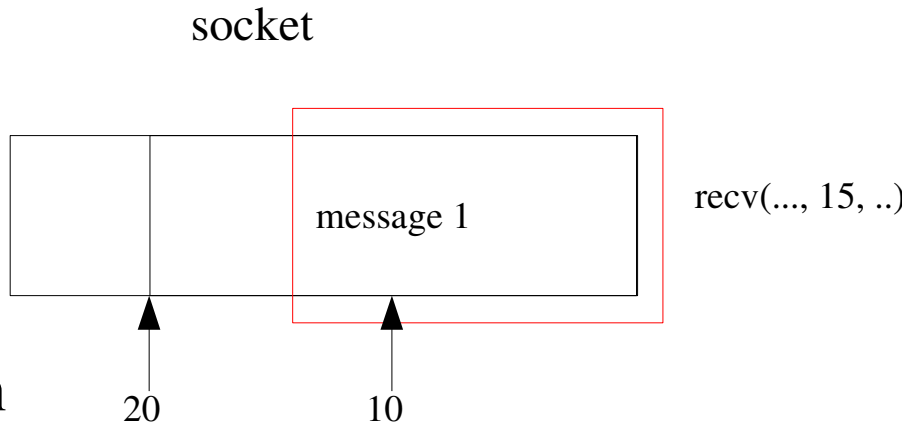
# Echange de données – mode connecté (1)

- cas analysé : réception d'un seul message en une seule fois
- réception de plusieurs messages
  - plusieurs envois consécutifs
  - pas de marque de séparation, structuration
    - réception pas forcément alignée sur les messages initiaux
    - réception de plusieurs messages en un seul



# Echange de données – mode connecté (2)

- réception d'une partie de message
  - taille des données inconnue
  - buffer de taille insuffisante
  - allocation dynamique
    - *malloc*
    - *realloc*
  - reconstituer le message
  - déterminer la fin de réception



# Fermeture – mode non-connecté

- fermeture de la socket (niveau système - fichiers)
- *int close(int sock)*
- fermeture par défaut en fin de programme
- nombre de descripteurs limité → libérer le descripteur de fichier
- faut-il toujours tester le retour d'erreur ?
  - oui, pour information

```
…
err = close(sock);
if (err < 0)
    //      erreur
…
```

# Fermeture – mode connecté

- socket + connexion = close +
  - fermeture de la connexion (niveau réseau)
  - *int shutdown(int sockfd, int how)* (sys/socket.h)
  - fermeture partielle ou totale *how*
    - *how = 0 (SHUT_RD)* : fermeture en lecture
    - *how = 1 (SHUT_WR)* : fermeture en écriture
    - *how = 2 (SHUT_RDWR)* : fermeture des deux
  - effet : complexe
    - *send*
      - si a fait le *shutdown* : signal SIGPIPE
      - sinon, rien pour le 1er message, puis SIGPIPE
    - *recv* = 0

# Exemple mode connecté – serveur

```
/* include généraux */
#include <stdio.h>
#include <stdlib.h>

/* fonctions des sockets TCP */
#include "fonctionsTCP.h"

/* constantes locales */
#define TAIL_BUF 100
#define SERV_PORT 2609

main(int argc, char** argv) {
   int sockCont, sockTrans, /* desc des sockets */
       err;                          /* code d'erreur */
   char  buffer[TAIL_BUF];      /* buffer de réception */

  /*  création de la socket, protocole TCP  */
   sockCont = socketServeur(SERV_PORT);
   if (sockCont < 0) {
     printf("serveur : erreur socketServeur\n");
     exit(2);
   }
```

```
   /* attente de connexion */
   sockTrans = accept(sockCont,  NULL, NULL);
   if (sockTrans < 0) {
      printf("serveur :  erreur sur accept");
      close(sockCont);
      exit(3);
   }
   /* réception et affichage du message du client */
   err = recv(sockTrans, buffer,TAIL_BUF, 0);
   if (err < 0) {
      printf("serveur : erreur dans la reception");
      shutdown(sockTrans, 2); close(sockTrans);
      close(sockCont);
      exit(4);
   }
   printf("serveur : le message recu: %s\n", buffer);

   /* arrêt de la connexion et fermeture  */
   shutdown(sockTrans, 2); close(sockTrans);
   close(sockCont);
}
```

# Exemple mode connecté – client

```
/* include généraux */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* fonctions des sockets TCP */
#include "fonctionsTCP.h"

/* constantes locales */
#define SERV_PORT 2609
#define TAIL_BUF 100

main(int argc, char **argv) {
  char chaine[TAIL_BUF];
  int sock,          /* descripteur de la socket locale */
      err;           /* code d'erreur */
  char* serveur = "localhost";
  /* création d'une socket client */
  sock = socketClient(serveur, SERV_PORT);
  if (sock < 0) {
    printf("client : erreur socketClient\n");
    exit(2);
  }
```

```
  /* saisie de la chaîne  */
  printf("client : donner une chaine : ");
  scanf("%s", chaine);

  /* envoi de la chaîne */
  err = send(sock, chaine,
                strlen(chaine)+1, 0);
  if (err != strlen(chaine)+1) {
     printf("client : erreur sur le send");
     shutdown(sock, 2); close(sock);
     exit(3);
  }
  printf("client : envoi realise\n");

  /* fermeture de la connexion et de la socket */
  shutdown(sock, 2);
  close(sock);
}
```

# Exemple mode connecté – compilation

- bibliothèque fonctions
  - fonctionsTCP.h
    - inclusions
    - prototypes fonctions
  - fonctionsTCP.c
    - code fonctions

```
#############
# fonctionsTCP
#############

# for Solaris
#LD_FLAGS = -lsocket -lnsl

# for Linux
LD_FLAGS =

all: client serveur

fonctionsTCP.o: fonctionsTCP.c fonctionsTCP.h
	gcc -c -o fonctionsTCP.o fonctionsTCP.c

client: client.c fonctionsTCP.o
	gcc client.c -o client fonctionsTCP.o  $(LD_FLAGS)

serveur: serveur.c fonctionsTCP.o
	gcc serveur.c -o serveur fonctionsTCP.o  $(LD_FLAGS)

clean:
	rm *~ ; rm -i \#* ; rm *.o; \
	rm client ; rm serveur
```

# Exemple mode non-connecté – émetteur

```
/* include généraux ... */
/* fonctions des sockets UDP */
#include "fonctionsUDP.h"
/* constantes locales */
#define EMET_PORT 2609
#define TAIL_BUF 100

main(int argc, char** argv) {
  int sock,              /* descripteur de la socket locale */
      err;               /* code d'erreur */
  char chaine[TAIL_BUF];
  char* nomMach = "localhost"; /* nom de la machine
                                  dest */
  int port = …                  /* saisie */
  struct sockaddr_in addrDest;  /* adresse du
                                  destinataire */
  /* création d'une socket non connectée */
  sock = socketUDP(EMET_PORT);
  if (sock < 0) {
    printf("emetteur : erreur %d de socketUDP \n",
                              sock);
    exit(2);
  }
  /* initialisation de l'adresse du destinataire */
  err = adresseUDP(nomMach, port, &addrDest);
  if (err  == -1) {
    printf("emetteur : erreur adresse\n");
    close(sock);    exit(3);
  }
  /*  saisie et envoi de la chaine  */
  printf("emetteur : donner la chaine : ");
  scanf("%s", chaine);
  err = sendto(sock, chaine, strlen(chaine)+1, 0,
               (struct sockaddr*)&addrDest,
               sizeof(struct sockaddr_in));
  if (err != strlen(chaine)+1) {
     printf("emetteur : erreur sur le send\n");
     close(sock);   exit(4);
  }
  printf("emetteur : chaine envoyee\n");
  /* fermeture de la socket */
  close(sock);
}
```

# Exemple mode non-connecté – récepteur

```
/* include généraux ...*/
/* fonctions des sockets UDP */
#include "fonctionsUDP.h"

/* constantes locales */
#define RECEPT_PORT 2610
#define TAIL_BUF 100

main(int argc, char** argv) {
   int  sock,      /* descripteur de socket locale */
        err;       /* code d'erreur */
   char buffer[TAIL_BUF]; /* buffer de réception*/

  /* création de la socket, protocole UDP */
   sock = socketUDP(RECEPT_PORT);
   if (sock < 0) {
     printf("recepteur : erreur %d socketUDP\n",
                                 sock);
     exit(2);
   }
```

```
   /* réception et affichage du message du client */
   err = recvfrom(sock, buffer, TAIL_BUF,
                          0,  NULL, NULL) ;
   if (err < 0) {
      printf("erreur dans la reception");
      close(sock);  exit(6);
   }

   printf("Voila le message recu: %s\n", buffer);

   /* fermeture de la socket */
   close(sock);
}
```

# Exemple mode non connecté – compilation

- bibliothèque fonctions :
  - fonctionsUDP.h
    - inclusions
    - prototypes fonctions
  - fonctionsUDP.c
    - code fonctions

```
#############
# fonctionsUDP
#############

# for Solaris
#LD_FLAGS = -lsocket -lnsl

# for Linux
LD_FLAGS =

all: emetteur recepteur

fonctionsUDP.o: fonctionsUDP.c fonctionsUDP.h
        gcc -c fonctionsUDP.c

emetteur: emetteur.c fonctionsUDP.o
        gcc emetteur.c -o emetteur fonctionsUDP.o

recepteur: recepteur.c fonctionsUDP.o
        gcc recepteur.c -o recepteur fonctionsUDP.o

clean:
        rm  *~ ; rm -i \#* ; rm *.o; \
        rm emetteur ; rm recepteur
```

# Recevoir l'adresse de l'émetteur (1)

- adresse socket
  - de la socket connectée : *int accept(int sockfd, NULL, NULL)*
  - de la socket émettrice : *ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, NULL, NULL)*
- deux derniers paramètres
  - *int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)*
  - *ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen)*

# Recevoir l'adresse de l'émetteur (2)

- retour de l'adresse par les fonctions
  - passage de paramètre par pointeur
    - structure adresse
    - taille adresse
  - une zone mémoire pour l'adresse : *struct sockaddr *addr*
  - donner la taille réservée : *socklen_t *addrlen*
- exemple

```
int tabInt[50];              // buffer de réception
sockaddr_in adEmet; // adresse de la socket de transmission
socklen_t   sizeAddr;  // taille de l'adresse
…
sizeAddr = sizeof(struct sockaddr_in);
…
// réception des données
err = recvfrom(sock, tabInt, sizeof(tabInt), 0,
                              (struct sockaddr*)&adEmet, &sizeAddr);
 if (err < 0)  // erreur
 …
```

# Gestion des erreurs (1)

- fonctions systèmes
  - toujours tester le retour
  - pas de garantie sur la disponibilité des ressources
- erreur
  - valeur de retour = -1
  - pas suffisamment précis
  - *errno*
    - *extern* (POSIX)
    - contient le code de l'erreur
    - *#include <errno.h>*

# Gestion des erreurs (2)

- exemple

```
#include <errno.h>

int main(...){
…
 // réception des données
 err = recvfrom(sock, buffer, TAIL_BUF, 0,
                        (struct sockaddr*)&adEmet, &sizeAddr);
if (err < 0)  // erreur
  printf("Recepteur : erreur sur recvfrom %d\n", errno);
```

- codes d'erreurs
  - *errno.h* : dépend du système
  - */usr/include/errno.h => /usr/include/asm-generic/errno.h*

```
#define EUSERS          87              /* Too many users */
#define ENOTSOCK        88              /* Socket operation on non-socket */
#define EDESTADDRREQ    89      /* Destination address required */
#define EMSGSIZE        90              /* Message too long */
#define EPROTOTYPE      91      /* Protocol wrong type for socket */
#define ENOPROTOOPT     92      /* Protocol not available */
#define EPROTONOSUPPORT 93      /* Protocol not supported */
#define ESOCKTNOSUPPORT 94      /* Socket type not supported */
#define EOPNOTSUPP      95      /* Operation not supported on transport endpoint*/
```

# Gestion des erreurs (3)

- autres utilisations
  - *void perror(const char *s)*
  - *char *strerror(int errnum)*

```
#include <stdio.h>

 int main(…) {
  …
  // réception des données
  err = recvfrom(sock, buffer, TAIL_BUF,
                       0, (struct sockaddr*)&adEmet,
                       &sizeAddr);
  if (err < 0)  // erreur
     perror("Recepteur : erreur sur recvfrom");
  …
```

```
#include <string.h>
#include <errno.h>


  int main(...){
   …
   // réception des données
   err = recvfrom(sock, buffer, TAIL_BUF, 0,
                        (struct sockaddr*)&adEmet,
                        &sizeAddr);
  if (err < 0) // erreur
    printf("Recepteur : erreur sur recvfrom %s\n",
                              strerror(errno));
  …
```

# Gestion des erreurs (4)

- Que faire en cas d'erreur ?
  - trace : information
  - sortir
    - erreur à la création de socket de connexion
    - *void exit(int status)*
      - 0 : OK
  - ne pas sortir : mauvaise réception = couper la connexion
- signal
  - *sighandler_t signal(int signum, sighandler_t handler)*
  - tue le processus par défaut (*SIGPIPE*)

# Divers (1)

- *connect* / *accept*
  - normalement *connect* attend *accept*
  - pas le cas sur Linux, pour améliorer performances (HTTP)
- fonctions d'accès fichiers, utilisables sur la socket
  - *ssize_t read(int fd, void *buf, size_t count)*
  - *ssize_t write(int fd, const void *buf, size_t count)*
  - autres fonctions : *fcntl*, *fflush*, etc.
- *recv* sur socket non-connectée = *recvfrom(..., NULL, NULL)*

# Divers (2)

- prototype des fonctions
  - anciennes définitions : *char**, *int*
  - pas identiques sur tous les systèmes
- communication des chaînes de caractères
  - *strlen* +1

# Les inclusions

- de base pour les sockets
  - *#include <sys/types.h>*
  - *#include <sys/socket.h>*
- Internet
  - *#include <netinet/in.h>*
  - *#include <netdb.h>*
- pour les fonctions socket SCS (inclure fonctions de gestion des sockets TCP/UDP)
  - #include "fonctionsSocket.h"
- fonctions utiles
  - *#include <stdlib.h>, #include <stdio.h>*
  - *#include <unistd.h>*
  - *#include <string.h>*