

HMIN105M : Principes de la programmation concurrente et répartie

Mise en œuvre d'applications distribuées

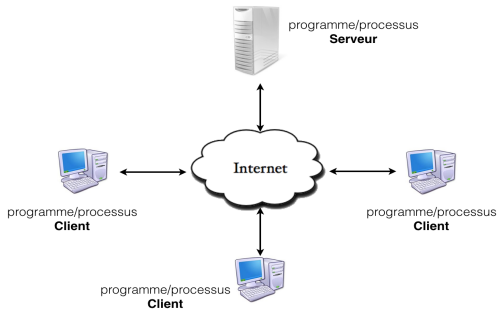
Responsable : Hinde Bouziane (bouziane@lirmm.fr)

Intervenants : H. Bouziane et T. Lorieul

UM - LIRMM

Objectif

Apprendre à programmer des applications nécessitant des communications distantes



En utilisant :

- les sockets
- deux protocoles : UDP et TCP
- le langage C - Noyau Unix

Le modèle client-serveur

Client :

processus qui envoie des requêtes au processus dit *serveur*, attend une réponse indiquant leurs réalisations et les éventuels résultats .

Le modèle client-serveur

Client :

processus qui envoie des requêtes au processus dit *serveur*, attend une réponse indiquant leurs réalisations et les éventuels résultats .

Serveur :

processus qui attend des requêtes provenant de processus clients, réalise ces requêtes et rend les résultats.

Le modèle client-serveur

Client :

processus qui envoie des requêtes au processus dit *serveur*, attend une réponse indiquant leurs réalisations et les éventuels résultats .

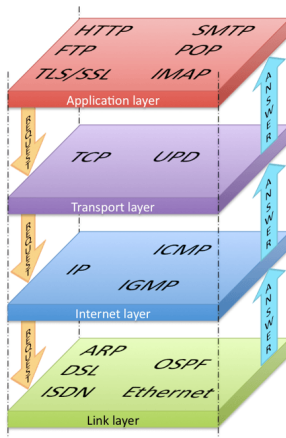
Serveur :

processus qui attend des requêtes provenant de processus clients, réalise ces requêtes et rend les résultats.

Requête :

suite d'instructions, commandes, ou simple chaîne de caractères, obéissant à un langage, un accord ou une structure préalables connus des deux entités communicantes (protocole d'application).

Positionnement OSI : Vision en 4 couches



Pour programmer une application client-serveur, dans quelle couche doit-on se positionner ? Quelle(s) autre(s) couche(s) doit-on utiliser ?

Contenu

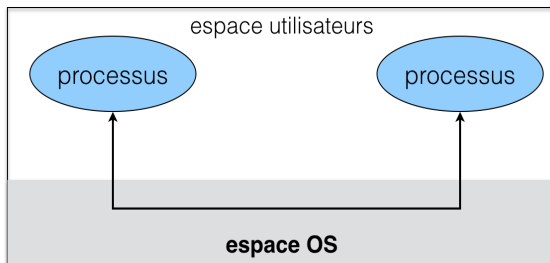
- 1 Introduction
- 2 Présentation des sockets
- 3 Communications en mode non connecté (UDP)
- 4 Communications en mode connecté (TCP)
- 5 Gestion de plusieurs clients (UDP et TCP)

- 1 Introduction
- 2 Présentation des sockets
- 3 Communications en mode non connecté (UDP)
- 4 Communications en mode connecté (TCP)
- 5 Gestion de plusieurs clients (UDP et TCP)

Communications dans les systèmes centralisés

Machine A

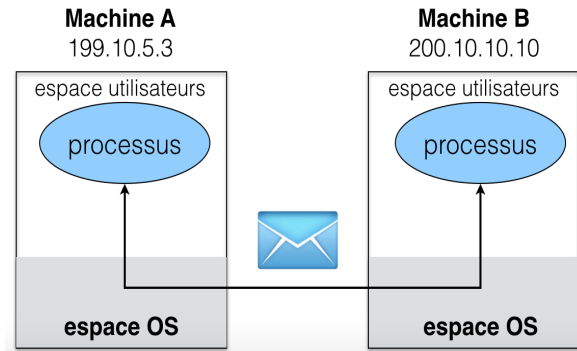
199.10.5.3



Pipes/tubes, autres ?

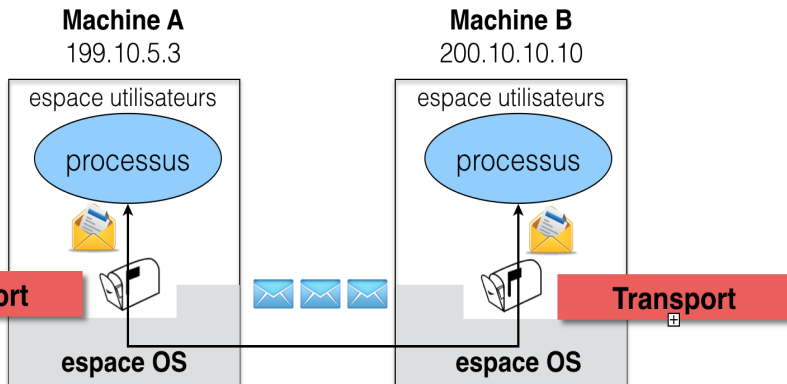
Supposent l'existence d'une mémoire partagée entre processus

Communications dans les systèmes répartis / distribués



Envoi / réception de messages

Communications dans les systèmes répartis / distribués



 ± **Socket + API**

- 1 Introduction
- 2 Présentation des sockets**
- 3 Communications en mode non connecté (UDP)
- 4 Communications en mode connecté (TCP)
- 5 Gestion de plusieurs clients (UDP et TCP)

Qu'est ce qu'une socket ?

Définition 1

Une socket (en français, prise) est une notion qui étend celle de tube / pipe. Elle permet de définir un canal de communication entre deux processus, sauf qu'elle permet en plus :

- la communication distante (en réseaux) ;
- le choix de différents protocoles de communication.

Qu'est ce qu'une socket ?

Définition 1

Une socket (en français, prise) est une notion qui étend celle de tube / pipe. Elle permet de définir un canal de communication entre deux processus, sauf qu'elle permet en plus :

- la communication distante (en réseaux) ;
- le choix de différents protocoles de communication.

Définition 2

Une socket est une extrémité d'un canal de communication bidirectionnel entre deux processus

Qu'est ce qu'une socket ?

Définition 1

Une socket (en français, prise) est une notion qui étend celle de tube / pipe. Elle permet de définir un canal de communication entre deux processus, sauf qu'elle permet en plus :

- la communication distante (en réseaux) ;
- le choix de différents protocoles de communication.

Définition 2

Une socket est une extrémité d'un canal de communication bidirectionnel entre deux processus

Bidirectionnel

Dans les deux sens : envoi et réception.

Propriétés d'une socket

Concrètement, une socket est définie par :

- un domaine d'appartenance
- un type
- un protocole
- une paire (adresse IP et numéro de port) pour la désigner dans le domaine.

Autres propriétés :

- une socket est identifiée par un descripteur de fichier
- une socket est associée à deux buffers :
 - de réception : contient les données reçues par la couche transport et à lire par la couche application,
 - d'émission : contient les données que la couche application transmet à la couche transport.
- Une socket est un concept multiplateforme et multi-langage.

Domaine de communication

Le domaine permet d'identifier une socket dans un domaine d'adresses et son utilisation dans une famille de protocoles connus.

Exemples :

- **IPv4 (PF_INET)** : L'adresse de la socket est une adresse IPv4 et la communication se fait avec un ou des processus distants suivant les protocoles Internet v4.
- **IPv6 (PF_INET6)** : L'adresse de la socket est une adresse IPv6 et la communication se fait avec un ou des processus distants suivant les protocoles Internet v6.
- **Unix (PF_UNIX)** : la communication ne peut se faire qu'entre processus se trouvant sur la même station Unix
- Etc. : voir la documentation

Nous utiliserons le domaine IPv4 (PF_INET).

Types d'une socket (1/3)

Le type d'une socket détermine le **format de transmission de données**, le **mode de connexion** utilisé avec un ou d'autres processus et des **propriétés de transmission** offertes par la couche transport.

Types d'une socket (1/3)

Le type d'une socket détermine le **format de transmission de données**, le **mode de connexion** utilisé avec un ou d'autres processus et des **propriétés de transmission** offertes par la couche transport.

Deux formats : datagramme et stream

- **Datagramme (SOCK_DGRAM)** : un message est expédié comme un paquet bien délimité et est reçu entièrement en une seule fois.
- **Stream (SOCK_STREAM)** : un message est expédié/reçu comme un flot continu de caractères. Si l'expéditeur envoie plusieurs messages, le destinataire pourra lire ces messages en une fois, en plusieurs fois ou caractère par caractère. Les limites des messages ne sont pas définies.

Types d'une socket (1/3)

Le type d'une socket détermine le **format de transmission de données**, le **mode de connexion** utilisé avec un ou d'autres processus et des **propriétés de transmission** offertes par la couche transport.

Deux formats : datagramme et stream

- **Datagramme (SOCK_DGRAM)** : un message est expédié comme un paquet bien délimité et est reçu entièrement en une seule fois.
- **Stream (SOCK_STREAM)** : un message est expédié/reçu comme un flot continu de caractères. Si l'expéditeur envoie plusieurs messages, le destinataire pourra lire ces messages en une fois, en plusieurs fois ou caractère par caractère. Les limites des messages ne sont pas définies.

Question : Dans le cas d'une socket SOCK_STREAM, à qui revient la responsabilité de définir les limites des messages ?

Types d'une socket (2/3)

Le type d'une socket détermine le format de transmission de données, le **mode de connexion** utilisé avec un ou d'autres processus et des propriétés de transmission offertes par la couche transport.

Deux modes de communication : connecté et non connecté

- **Connecté** : la transmission de messages est précédée par une phase de connexion avec une autre socket. Une socket en mode connecté est donc utilisée pour communiquer de façon exclusive avec une seule autre socket. Nous parlons de *circuit ou canal virtuel* établi entre les deux sockets (analogie : communications téléphoniques)
- **Non connecté** : la destination d'un message à émettre via une socket en mode non connectée n'est pas nécessairement la même que celle du message suivant. A chaque émission, une adresse de destination doit être spécifiée (analogie : communications par courrier postal)

Types d'une socket (3/3)

Le type d'une socket détermine le format de transmission de données, le mode de connexion utilisé avec un ou d'autres processus et des **propriétés de transmission** offertes par la couche transport.

Propriétés des transmissions

- Fiabilité : *soit un message transmis arrive bien à destination, soit une erreur est retournée à l'application.*
- Paquets/messages non dupliqués
- Remise dans l'ordre des paquets : *si des paquets sont reçus dans le désordre, la couche transport prendrait en charge la remise en ordre avant le dépôt d'un message à l'application.*

Types d'une socket (3/3)

Le type d'une socket détermine le format de transmission de données, le mode de connexion utilisé avec un ou d'autres processus et des **propriétés de transmission** offertes par la couche transport.

Propriétés des transmissions

- Fiabilité : *soit un message transmis arrive bien à destination, soit une erreur est retournée à l'application.*
- Paquets/messages non dupliqués
- Remise dans l'ordre des paquets : *si des paquets sont reçus dans le désordre, la couche transport prendrait en charge la remise en ordre avant le dépôt d'un message à l'application.*

Remarque : la garantie de ces propriétés dépend des protocoles utilisés.

Protocoles

Par défaut :

- Une socket de type datagramme (SOCK_DGRAM) utilise le protocole **UDP** (User Datagram Protocol)
- Une socket de type stream (SOCK_STREAM) utilise le protocole **TCP** (Transmission Control Protocol)

TCP	UDP
fiable ordre garanti duplication impossible mode connecté	non fiable ordre non garanti duplication possible mode non connecté

Il existe d'autres protocoles que vous pouvez trouver dans la documentation.

Désigner une socket dans un domaine : Nommage

- Pour pouvoir se connecter ou envoyer des données à une socket distante, il est nécessaire de connaître son adresse (adresse IP, numéro de port).
- L'association d'une adresse à une socket est appelée : nommage.
- Deux méthodes de nommage :
 - Le programmeur ou l'utilisateur choisi une adresse IP et un numéro de port. Dans ce cas, le système vérifie leur disponibilité.
 - Laisser le système choisir une ou des adresses. Il est possible ensuite de consulter ces informations pour les connaître.

Question : dans une application client-serveur, doit-on connaître les adresses de tous les processus clients et processus serveurs pour pouvoir communiquer ?

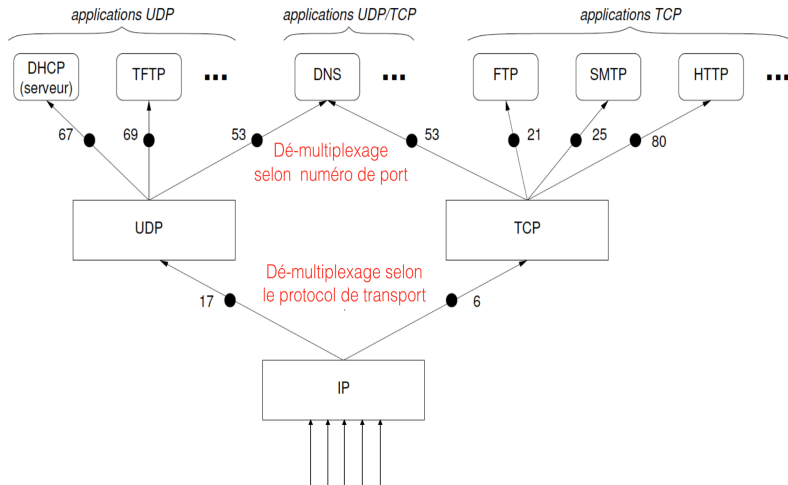
Allocation des numéros de port (1/2)

- Dans l'Internet, chaque application client-serveur va se voir attribuer un ou des numéros de port **public**(s).
- Les numéros jusqu'à 1024 sont officiellement réservés pour des applications connues et ne peuvent être demandés par une application d'utilisateur non administrateur.
- Exemples :
 - Tous les serveurs `sshd` utilisent strictement le port numéro 22 ;
 - Tous les serveurs `httpd` utilisent *par défaut* le port numéro 80.
- Un client peut localiser les serveurs connus, dès lors que ces numéros de port sont enregistrés dans un fichier local (voir sous Unix `/etc/services`).
- Pour vos programmes, les numéros de ports seront supérieurs à 1024.

Allocation des numéros de port (2/2)

- Deux sockets sur une même station, peuvent avoir un même numéro de port.
- En effet, chaque protocole de transport a sa propre numérotation des sockets. Donc un numéro de port n'indique pas de quel protocole il s'agit.
- Un programme/processus communiquant, peut utiliser plusieurs sockets utilisant un ou différents protocoles.
- Pour réaliser une communication avec un processus, il faut identifier la socket à utiliser.

Démultiplexage des ports



En pratique ?

Utilisation d'une API pour :

- créer une socket
- nommer une socket
- se connecter à une socket distante
- recevoir un message
- envoyer un message
- fermer une socket

La connexion de deux sockets ainsi que la réception et l'envoi de messages seront présentés en fonction du protocole utilisé (UDP ou TCP)

Création d'une socket

```
int socket(int domain, int type, int protocole)
```

Création d'une socket

```
int socket(int domain, int type, int protocole)
```

Exemple 1

```
int dSock = socket(PF_INET, SOCK_DGRAM, 0);
```

crée une socket dans le domaine IPv4, de type datagramme et utilisant le protocole par défaut : ici UDP.

Création d'une socket

```
int socket(int domain, int type, int protocole)
```

Exemple 1

```
int dSock = socket(PF_INET, SOCK_DGRAM, 0);
```

crée une socket dans le domaine IPv4, de type datagramme et utilisant le protocole par défaut : ici UDP.

Exemple 2

```
int dSock = socket(PF_INET, SOCK_STREAM, 0);
```

Que fait cet appel ?

Création d'une socket

```
int socket(int domain, int type, int protocole)
```

Exemple 1

```
int dSock = socket(PF_INET, SOCK_DGRAM, 0);
```

créé une socket dans le domaine IPv4, de type datagramme et utilisant le protocole par défaut : ici UDP.

Exemple 2

```
int dSock = socket(PF_INET, SOCK_STREAM, 0);
```

Que fait cet appel ?

Valeur de retour

Le descripteur de la socket créée ou -1 en cas d'erreur.

Nommage d'une socket (1/3)

```
int bind (int descripteur,           // descripteur de socket  
          const struct sockaddr *adresse, // pointeur vers l'adresse  
          socklen_t lgAdr)           // longueur de l'adresse
```

Nommage d'une socket (1/3)

```
int bind (int descripteur,           // descripteur de socket  
          const struct sockaddr *adresse, // pointeur vers l'adresse  
          socklen_t lgAdr)           // longueur de l'adresse
```

La structure sockaddr

```
struct sockaddr {  
    sa_family_t sa_family; // famille d'adresses, AF_XXX  
    char sa_data[14]; // 14 octets pour l'IP + port  
};
```

Elle définit un type générique d'adresses. Le type réellement utilisé varie en fonction du domaine de la socket :

- Si `PF_INET` alors `struct sockaddr_in` : une adresse IPv4 et un port
- Si `PF_INET6` alors `struct sockaddr_in6` : une adresse IPv6 et un port

Nommage d'une socket (2/3)

La structure sockaddr_in

```
struct sockaddr_in {  
    sa_family_t sin_family;  // famille AF_INET  
    in_port_t sin_port;     // numéro de port au format réseau  
    struct in_addr sin_addr; // structure d'adresse IP  
};  
struct in_addr {  uint32_t s_addr; // adresse IP au format réseau };
```

Remarques

- L'IP et le numéro de port sont stockés au format réseau (network byte order) : XXxx
- Les entiers sont au format hôte (host byte order) : xxXX ou XXxx
- Une conversion est nécessaire : fonctions ntohs(), htons(), ntohl(), htonl()

Nommage d'une socket (3/3)

Exemple : que fait le code suivant ?

```
int dSock = socket(PF_INET, SOCK_STREAM, 0);  
struct sockaddr_in ad;  
ad.sin_family = AF_INET;  
ad.sin_addr.s_addr = INADDR_ANY;  
ad.sin_port = htons((short)31470);  
int res = bind(dSock, (struct sockaddr*)&ad, sizeof(ad));
```

INADDR_ANY

Attache la socket à toutes les interfaces réseaux locales.

Valeur de retour

0 si le nommage a réussi, -1 sinon (e.g. si le port est déjà utilisé).

Fermeture d'une socket

```
int close (int descripteur);  
int shutdown(int descripteur, int comment);  
           // comment : SHUT_WR arrêt émission ou  
           // SHUT_RDWR pour l'émission et la réception
```

Valeur de retour

0 si la fermeture a réussi, -1 sinon.

- 1 Introduction
- 2 Présentation des sockets
- 3 Communications en mode non connecté (UDP)**
- 4 Communications en mode connecté (TCP)
- 5 Gestion de plusieurs clients (UDP et TCP)

Principe (illustré avec deux programmes)

Programme 1

Programme 2

Principe (illustré avec deux programmes)

Programme 1

créer une socket

Programme 2

créer une socket

Principe (illustré avec deux programmes)

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Principe (illustré avec deux programmes)

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

Principe (illustré avec deux programmes)

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

```
bind(dS, ....);
```

Principe (illustré avec deux programmes)

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Communiquer

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

bind(dS,);

Communiquer

Principe (illustré avec deux programmes)

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Communiquer

```
sendto(dS, ...);
```

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

```
bind(dS, ....);
```

Communiquer

```
recvfrom(dS, ...);
```

Principe (illustré avec deux programmes)

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Communiquer

```
sendto(dS, ...);  
recvfrom(dS, ...);
```

...

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

```
bind(dS, ....);
```

Communiquer

```
recvfrom(dS, ...);  
sendto(dS, ...);
```

...

Principe (illustré avec deux programmes)

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Communiquer

```
sendto(dS, ...);
```

```
recvfrom(dS, ...);
```

...

Fermer la Socket

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

```
bind(dS, ....);
```

Communiquer

```
recvfrom(dS, ...);
```

```
sendto(dS, ...);
```

...

Fermer la socket

Principe (illustré avec deux programmes)

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Communiquer

```
sendto(dS, ...);  
recvfrom(dS, ...);
```

...

Fermer la Socket

```
close(dS);
```

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

```
bind(dS, ....);
```

Communiquer

```
recvfrom(dS, ...);  
sendto(dS, ...);
```

...

Fermer la socket

```
close(dS);
```

Principe (illustré avec deux programmes)

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Communiquer

```
sendto(dS, ...);  
recvfrom(dS, ...);
```

...

Fermer la Socket

```
close(dS);
```

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

```
bind(dS, ....);
```

Communiquer

```
recvfrom(dS, ...);  
sendto(dS, ...);
```

...

Fermer la socket

```
close(dS);
```

En supposant que :

- Programme 1 connaît l'adresse de la socket du Programme 2, mais pas l'inverse : *Nous sommes en **mode asymétrique***

Principe - suite : mode symétrique

Programme 1

Programme 2

Principe - suite : mode symétrique

Programme 1

créer une socket

Programme 2

créer une socket

Principe - suite : mode symétrique

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Principe - suite : mode symétrique

Programme 1

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

Programme 2

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

Principe - suite : mode symétrique

Programme 1

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

bind(dS,);

Programme 2

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

bind(dS,);

Principe - suite : mode symétrique

Programme 1

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

bind(dS,);

Communiquer

Programme 2

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

bind(dS,);

Communiquer

Principe - suite : mode symétrique

Programme 1

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

bind(dS,);

Communiquer

sendto(dS, ...);

Programme 2

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

bind(dS,);

Communiquer

sendto(dS, ...);

Principe - suite : mode symétrique

Programme 1

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

bind(dS,);

Communiquer

sendto(dS, ...);

recvfrom(dS, ...);

...

Programme 2

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

bind(dS,);

Communiquer

sendto(dS, ...);

recvfrom(dS, ...);

...

Principe - suite : mode symétrique

Programme 1

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

bind(dS,);

Communiquer

sendto(dS, ...);

recvfrom(dS, ...);

...

Fermer la Socket

Programme 2

créer une socket

dS = **socket**(..., SOCK_DGRAM, ...);

Nommer la socket

bind(dS,);

Communiquer

sendto(dS, ...);

recvfrom(dS, ...);

...

Fermer la socket

Principe - suite : mode symétrique

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

```
bind(dS, ....);
```

Communiquer

```
sendto(dS, ...);
```

```
recvfrom(dS, ...);
```

...

Fermer la Socket

```
close(dS);
```

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

```
bind(dS, ....);
```

Communiquer

```
sendto(dS, ...);
```

```
recvfrom(dS, ...);
```

...

Fermer la socket

```
close(dS);
```

Principe - suite : mode symétrique

Programme 1

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

```
bind(dS, ....);
```

Communiquer

```
sendto(dS, ...);
```

```
recvfrom(dS, ...);
```

...

Fermer la Socket

```
close(dS);
```

Programme 2

créer une socket

```
dS = socket(..., SOCK_DGRAM, ...);
```

Nommer la socket

```
bind(dS, ....);
```

Communiquer

```
sendto(dS, ...);
```

```
recvfrom(dS, ...);
```

...

Fermer la socket

```
close(dS);
```

Dans le mode symétrique :

- Chaque programme connaît l'adresse de la socket de l'autre programme avant de communiquer.

Remarques

- Le principe d'utilisation du protocole UDP a été illustré avec deux processus.
- Ce principe est extensible à plusieurs processus.
- Avant d'envoyer un message vers une socket distante, veiller à ce que le nommage de cette socket soit déjà fait. Que se passe t-il sinon ?
- *L'ordre dans lequel les envois et réceptions sont à réaliser est propre à chaque application. Ce cours ne donne que des principes de base et ne parcourt pas toutes les possibilités, donc évitez des réponses : "j'ai fait ainsi car c'est dans le cours !"*
- Attention aux situations d'interblocage. Donnez un exemple.

Réception d'un message

```
ssize_t recvfrom (int descripteur, // descripteur de socket
                  const void *msg, // * pointeur vers le premier octet où
                                   // sera stocké le message reçu */
                  size_t lg, // le nombre max d'octets attendus
                  int flags, // options de réception, 0 par défaut
                  const struct sockaddr *adrExp, // pointeur vers l'adresse
                                                  // de la socket de l'expéditeur
                  socklen_t lgAdr) // longueur de l'adresse
```

Réception d'un message

```

ssize_t recvfrom (int descripteur,           // descripteur de socket
                  const void *msg,           /* pointeur vers le premier octet où
                                              sera stocké le message reçu */
                  size_t lg,                 // le nombre max d'octets attendus
                  int flags,                 // options de réception, 0 par défaut
                  const struct sockaddr *adrExp, // pointeur vers l'adresse
                                              // de la socket de l'expéditeur
                  socklen_t lgAdr)           // longueur de l'adresse

```

Valeur de retour

*En UDP, si l'appel réussit, le nombre d'octets effectivement extraits ($\leq lg$), -1 sinon (avec *errno* positionné).*

Réception d'un message

```

ssize_t recvfrom (int descripteur,           // descripteur de socket
                  const void *msg,           /* pointeur vers le premier octet où
                                              sera stocké le message reçu */
                  size_t lg,                 // le nombre max d'octets attendus
                  int flags,                 // options de réception, 0 par défaut
                  const struct sockaddr *adrExp, // pointeur vers l'adresse
                                              // de la socket de l'expéditeur
                  socklen_t lgAdr)           // longueur de l'adresse

```

Valeur de retour

En UDP, si l'appel réussit, le nombre d'octets effectivement extraits ($\leq lg$), -1 sinon (avec `errno` positionné).

Note

Cette fonction est bloquante. Dans quel cas ?

Réception d'un message - exemple

Prérequis

- Une socket correctement créée
- *Espace mémoire alloué (statiquement ou dynamiquement) pour stocker le message reçu et l'adresse de l'expéditeur.*

Réception d'un message - exemple

Prérequis

- Une socket correctement créée
- *Espace mémoire alloué (statiquement ou dynamiquement) pour stocker le message reçu et l'adresse de l'expéditeur.*

Que fait le code suivant ?

```
... // code incluant la création d'une socket (descripteur dSock)
struct sockaddr_in adExp;
socklen_t lgAdr = sizeof(struct sockaddr_in);
int message [10];
ssize_t res = recvfrom(dSock, message, sizeof(message), 0, &adExp,
&lgAdr);
```

Réception d'un message - exemple

Prérequis

- Une socket correctement créée
- *Espace mémoire alloué (statiquement ou dynamiquement) pour stocker le message reçu et l'adresse de l'expéditeur.*

Que fait le code suivant ?

```
... // code incluant la création d'une socket (descripteur dSock)
struct sockaddr_in adExp;
socklen_t lgAdr = sizeof(struct sockaddr_in);
int message [10];
ssize_t res = recvfrom(dSock, message, sizeof(message), 0, &adExp,
&lgAdr);
```

Et que se passe t-il si on avait

```
ssize_t res = recvfrom(dSock, message, sizeof(message), 0, NULL, NULL);
```

Envoi d'un message

```
ssize_t sendto (int descripteur,           // descripteur de socket  
                const void *msg,           /* pointeur vers le premier octet  
                                           du message à envoyer */  
                size_t lg,                 // le nombre d'octets du message  
                int flags,                  // options d'envoi, 0 par défaut  
                const struct sockaddr *adrDest, /* pointeur vers l'adresse  
                                              de la socket du destinataire */  
                socklen_t lgAdr)           // longueur de l'adresse
```

Envoi d'un message

```

ssize_t sendto (int descripteur,           // descripteur de socket
                 const void *msg,          /* pointeur vers le premier octet
                                           du message à envoyer */

                 size_t lg,                // le nombre d'octets du message
                 int flags,                // options d'envoi, 0 par défaut
                 const struct sockaddr *adrDest, /* pointeur vers l'adresse
                                           de la socket du destinataire */
                 socklen_t lgAdr)          // longueur de l'adresse

```

Valeur de retour

En UDP, si l'appel réussit, le nombre d'octets effectivement déposés dans le buffer de la socket, -1 sinon (avec errno positionné).

Envoi d'un message - exemples

Prérequis

- Une socket correctement créée et le message à envoyer bien construit.
- Connaître l'adresse de la socket du destinataire. Comment ?
 - *obtenue lors de la réception d'un message (mode asymétrique)*
 - *autrement : passée en paramètre, saisie, etc. (mode (a)symétrique)*

Envoi d'un message - exemples

Prérequis

- Une socket correctement créée et le message à envoyer bien construit.
- Connaître l'adresse de la socket du destinataire. Comment ?
 - *obtenue lors de la réception d'un message (mode asymétrique)*
 - *autrement : passée en paramètre, saisie, etc. (mode (a)symétrique)*

Exemple 1 : que fait le code suivant ?

```
... // code incluant la création d'une socket (descripteur dSock)
struct sockaddr_in adExp;
socklen_t lgAdr = sizeof(struct sockaddr_in);
int msg [10];
ssize_t res = recvfrom(dSock, msg, sizeof(msg), 0, &adExp, &lgAdr);
char* reponse = "Merci" ;
ssize_t res = sendto (dSock, reponse, 5, 0, (sockaddr *)adExp, lgAdr);
```


Envoi d'un message - exemples

Prérequis

- Une socket correctement créée et le message à envoyer bien construit.
- Connaître l'adresse de la socket du destinataire. Comment ?
 - *obtenue lors de la réception d'un message (mode asymétrique)*
 - *autrement : passée en paramètre, saisie, etc. (mode (a)symétrique)*

Envoi d'un message - exemples

Prérequis

- Une socket correctement créée et le message à envoyer bien construit.
- Connaître l'adresse de la socket du destinataire. Comment ?
 - *obtenue lors de la réception d'un message (mode asymétrique)*
 - *autrement : passée en paramètre, saisie, etc. (mode (a)symétrique)*

Exemple 2 : que fait le code suivant ?

```
... // code incluant la création d'une socket (descripteur dSock)
struct sockaddr_in adDest ;
adDest.sin_family = AF_INET ;
adDest.sin_port = htons(atoi(argv[2])) ;
int res = inet_pton(AF_INET, argv[1], &(adDest.sin_addr)) ;
socklen_t lgAdr = sizeof(struct sockaddr_in) ;
char msg = 'A' ;
res = sendto(dSock, &msg, 1, 0, (sockaddr *)adDest, lgAdr) ;
```

On reprend tout : exemple

Prog 1 (extrait)

```
dS= socket(PF_INET, SOCK_DGRAM, 0);
struct sockaddr_in aD;
aD.sin_family = AF_INET;
inet_pton(AF_INET,argv[1],&(aD.sin_addr));
aD.sin_port = htons(atoi(argv[2]));
socklen_t lgA = sizeof(struct sockaddr_in);
char * m = "Bonjour";
sendto(dS, m, 8, 0, (sockaddr*)&aD, lgA);
int r;
recvfrom(dS, &r, sizeof(int),0,NULL,NULL);
printf("reponse : %d", r);
close (dS);
```

Prog 2 (extrait)

```
dS= socket(PF_INET, SOCK_DGRAM, 0);
struct sockaddr_in ad;
ad.sin_family = AF_INET;
ad.sin_addr.s_addr = INADDR_ANY;
ad.sin_port = htons(atoi(argv[1]));
bind(dS, (struct sockaddr*)&ad, sizeof(ad));
struct sockaddr_in aE;
socklen_t lg = sizeof(struct sockaddr_in);
char m [20];
recvfrom(dS, m, sizeof(m), 0, (sockaddr*)&aE, &lg);
printf("recu : %s", m);
int r = 10;
sendto(dS, &r, sizeof(int),0,(sockaddr*)&aE,lg);
close (dS);
```

On reprend tout : exemple

Prog 1 (extrait)

```
dS= socket(PF_INET, SOCK_DGRAM, 0);
struct sockaddr_in aD;
aD.sin_family = AF_INET;
inet_pton(AF_INET,argv[1],&(aD.sin_addr));
aD.sin_port = htons(atoi(argv[2]));
socklen_t lgA = sizeof(struct sockaddr_in);
char * m = "Bonjour";
sendto(dS, m, 8, 0, (sockaddr*)&aD, lgA);
int r;
recvfrom(dS, &r, sizeof(int),0,NULL,NULL);
printf("reponse : %d", r);
close (dS);
```

Prog 2 (extrait)

```
dS= socket(PF_INET, SOCK_DGRAM, 0);
struct sockaddr_in ad;
ad.sin_family = AF_INET;
ad.sin_addr.s_addr = INADDR_ANY;
ad.sin_port = htons(atoi(argv[1]));
bind(dS, (struct sockaddr*)&ad, sizeof(ad));
struct sockaddr_in aE;
socklen_t lg = sizeof(struct sockaddr_in);
char m [20];
recvfrom(dS, m, sizeof(m), 0, (sockaddr*)&aE, &lg);
printf("recu : %s", m);
int r = 10;
sendto(dS, &r, sizeof(int),0,(sockaddr*)&aE,lg);
close (dS);
```

Question 1 :

Prog 2 envoie un message à Prog 1, pourtant, Prog 1 ne nomme pas sa socket. Comment est ce possible ?

On reprend tout : exemple

Prog 1 (extrait)

```
dS= socket(PF_INET, SOCK_DGRAM, 0);
struct sockaddr_in aD;
aD.sin_family = AF_INET;
inet_pton(AF_INET,argv[1],&(aD.sin_addr));
aD.sin_port = htons(atoi(argv[2]));
socklen_t lgA = sizeof(struct sockaddr_in);
char * m = "Bonjour";
sendto(dS, m, 8, 0, (sockaddr*)aD, lgA);
int r;
recvfrom(dS, &r, sizeof(int),0,NULL,NULL);
printf("reponse : %d", r);
close (dS);
```

Prog 2 (extrait)

```
dS= socket(PF_INET, SOCK_DGRAM, 0);
struct sockaddr_in ad;
ad.sin_family = AF_INET;
ad.sin_addr.s_addr = INADDR_ANY;
ad.sin_port = htons(atoi(argv[1]));
bind(dS, (struct sockaddr*)&ad, sizeof(ad));
struct sockaddr_in aE;
socklen_t lg = sizeof(struct sockaddr_in);
char m [20];
recvfrom(dS, m, sizeof(m), 0, &aE, &lg);
printf("recu : %s", m);
int r = 10;
sendto(dS, &r, sizeof(int),0,(sockaddr*)aE,lg);
close (dS);
```

Question 2 :

Que se passe-t-il si l'envoi du message dans Prog 1 se fait avant l'appel à bind() dans Prog2 ? Comment résoudre ce problème ?

Pour finir

- Un message en UDP est vu comme *un paquet qui peut être reçu et envoyé en une seule fois.*
- Il n'est pas nécessaire d'être en réception pour recevoir un message.
- *Un protocole d'application (échange) est indispensable pour une interprétation correcte des messages échangés.*
- UDP est un protocole non fiable. Il peut être nécessaire de s'assurer du bon fonctionnement des applications au niveau applicatif (exemple : s'assurer qu'une socket destinataire existe avant d'envoyer un message).
- Le code présenté est simplifié. *Il est indispensable de tester les valeurs de retour et de gérer les erreurs.*

- 1 Introduction
- 2 Présentation des sockets
- 3 Communications en mode non connecté (UDP)
- 4 Communications en mode connecté (TCP)**
- 5 Gestion de plusieurs clients (UDP et TCP)

Principe (illustré avec deux programmes)

Client

Serveur

Principe (illustré avec deux programmes)

Client

Créer une socket

Serveur

Créer une socket

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

```
bind(dS, ....);
```

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

```
bind(dS, ....);
```

Passer la socket en mode écoute

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

```
bind(dS, ....);
```

Passer la socket en mode écoute

```
listen(dS, ....);
```

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Demander une connexion

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

```
bind(dS, ....);
```

Passer la socket en mode écoute

```
listen(dS, ....);
```

Accepter une connexion

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Demander une connexion

```
connect(dS, ...);
```

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

```
bind(dS, ....);
```

Passer la socket en mode écoute

```
listen(dS, ....);
```

Accepter une connexion

```
dSClient = accept(dS, ...);
```


Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Demander une connexion

```
connect(dS, ...);
```

Communiquer

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

```
bind(dS, ....);
```

Passer la socket en mode écoute

```
listen(dS, ....);
```

Accepter une connexion

```
dSClient = accept(dS, ...);
```

Communiquer

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Demander une connexion

```
connect(dS, ...);
```

Communiquer

```
send(dS, ...);
```

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

```
bind(dS, ....);
```

Passer la socket en mode écoute

```
listen(dS, ....);
```

Accepter une connexion

```
dSClient = accept(dS, ...);
```

Communiquer

```
recv(dSClient, ...);
```

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Demander une connexion

```
connect(dS, ...);
```

Communiquer

```
send(dS, ...);
```

```
recv(dS, ...);
```

...

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

```
bind(dS, ....);
```

Passer la socket en mode écoute

```
listen(dS, ....);
```

Accepter une connexion

```
dSClient = accept(dS, ...);
```

Communiquer

```
recv(dSClient, ...);
```

```
send(dSClient, ...);
```

...

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Demander une connexion

```
connect(dS, ...);
```

Communiquer

```
send(dS, ...);
```

```
recv(dS, ...);
```

...

Fermer la Socket

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

```
bind(dS, ....);
```

Passer la socket en mode écoute

```
listen(dS, ....);
```

Accepter une connexion

```
dSClient = accept(dS, ...);
```

Communiquer

```
recv(dSClient, ...);
```

```
send(dSClient, ...);
```

...

Fermer les sockets

Principe (illustré avec deux programmes)

Client

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Demander une connexion

```
connect(dS, ...);
```

Communiquer

```
send(dS, ...);
```

```
recv(dS, ...);
```

...

Fermer la Socket

```
close(dS);
```

Serveur

Créer une socket

```
dS = socket(.., SOCK_STREAM, ..);
```

Nommer la socket

```
bind(dS, ....);
```

Passer la socket en mode écoute

```
listen(dS, ....);
```

Accepter une connexion

```
dSClient = accept(dS, ...);
```

Communiquer

```
recv(dSClient, ...);
```

```
send(dSClient, ...);
```

...

Fermer les sockets

```
close(dSClient); close(dS);
```

Passer une socket en mode écoute

```
int listen(int descripteur, int nbMaxEnAttente)
```

Passer une socket en mode écoute

```
int listen(int descripteur, int nbMaxEnAttente)
```

Exemple

```
... // code incluant la création d'une socket (descripteur dS)  
int res = listen(dS, 10);
```

Passes la socket dont le descripteur est dS en écoute de demandes de connexion. 10 est le nombre maximum de demandes de connexion pouvant être mises en attente (fixe une longueur de file d'attente).

Passer une socket en mode écoute

```
int listen(int descripteur, int nbMaxEnAttente)
```

Exemple

```
... // code incluant la création d'une socket (descripteur dS)  
int res = listen(dS, 10);
```

Passer la socket dont le descripteur est dS en écoute de demandes de connexion. 10 est le nombre maximum de demandes de connexion pouvant être mises en attente (fixe une longueur de file d'attente).

Valeur de retour

0 en cas de succès, -1 sinon (avec errno positionné).

Demande de connexion à un serveur

```
int connect(int descr, const struct sockaddr *adServ, socklen_t lgAdr)
```

Demande de connexion à un serveur

```
int connect(int descr, const struct sockaddr *adServ, socklen_t lgAdr)
```

Exemple

```
... // code incluant la création d'une socket (descripteur dS)
struct sockaddr_in adServ;
adServ.sin_family = AF_INET; adServ.sin_port = htons(34567);
int res = inet_pton(AF_INET, "197.50.51.10", &(adServ.sin_addr));
socklen_t lgA = sizeof(struct sockaddr_in);
res = connect(dS, (struct sockaddr *) &adServ, lgA);
```

Envoie une demande de connexion de la socket du client dS au serveur, via la socket du serveur dont l'IP est 197.50.51.10 et le numéro de port 34567.

Note : *favorisez le passage de paramètres ou la saisie des IP et numéro de port.*

Demande de connexion à un serveur

```
int connect(int descr, const struct sockaddr *adServ, socklen_t lgAdr)
```

Exemple

```
... // code incluant la création d'une socket (descripteur dS)
struct sockaddr_in adServ;
adServ.sin_family = AF_INET; adServ.sin_port = htons(34567);
int res = inet_pton(AF_INET, "197.50.51.10", &(adServ.sin_addr));
socklen_t lgA = sizeof(struct sockaddr_in);
res = connect(dS, (struct sockaddr *) &adServ, lgA);
```

Envoie une demande de connexion de la socket du client dS au serveur, via la socket du serveur dont l'IP est 197.50.51.10 et le numéro de port 34567.

Note : *favorisez le passage de paramètres ou la saisie des IP et numéro de port.*

Valeur de retour

0 en cas de succès, -1 sinon (avec errno positionné).

Accepter une demande de connexion d'un client

```
int accept(int descr,          /* descripteur de la socket recevant des  
                                demandes de connexion */  
    struct sockaddr *adrClient, // pointeur vers l'adresse de la socket du client  
    socklen_t lgAdr)           // pointeur vers longueur de l'adresse
```

Accepter une demande de connexion d'un client

```
int accept(int descr,          /* descripteur de la socket recevant des
                               demandes de connexion */
           struct sockaddr *adrClient, // pointeur vers l'adresse de la socket du client
           socklen_t lgAdr)      // pointeur vers longueur de l'adresse
```

Exemple

```
... // code incluant la création d'une socket (descripteur dS)
int res = listen(dS, 10);
struct sockaddr_in adClient; socklen_t lgA = sizeof(struct sockaddr_in);
int dSClient = accept(dS, (struct sockaddr *) &adClient, &lgA);
```

Extrait une demande de connexion de la socket dS et la traite. En cas de succès, une nouvelle socket (descripteur dSClient) est créée et connectée à la socket du client demandeur et dont l'adresse est stockée dans adClient

Accepter une demande de connexion d'un client

```
int accept(int descr,          /* descripteur de la socket recevant des  
                               demandes de connexion */  
           struct sockaddr *adrClient, // pointeur vers l'adresse de la socket du client  
           socklen_t lgAdr)         // pointeur vers longueur de l'adresse
```

Exemple

```
... // code incluant la création d'une socket (descripteur dS)  
int res = listen(dS, 10);  
struct sockaddr_in adClient; socklen_t lgA = sizeof(struct sockaddr_in);  
int dSClient = accept(dS, (struct sockaddr *) &adClient, &lgA);
```

Extrait une demande de connexion de la socket dS et la traite. En cas de succès, une nouvelle socket (descripteur dSClient) est créée et connectée à la socket du client demandeur et dont l'adresse est stockée dans adClient

Valeur de retour

Si demande acceptée, le descripteur (> 0) de la socket créée, -1 sinon (avec errno positionné).

Réception d'un message

```
ssize_t recv (int descripteur, // descripteur de socket  
              const void *msg, // * pointeur vers le premier octet où  
                               // sera stocké le message reçu */  
              size_t lg,       // le nombre max d'octets attendus  
              int flags)       // options de réception, 0 par défaut
```

Réception d'un message

```
ssize_t recv (int descripteur, // descripteur de socket  
              const void *msg, // * pointeur vers le premier octet où  
                               // sera stocké le message reçu */  
              size_t lg, // le nombre max d'octets attendus  
              int flags) // options de réception, 0 par défaut
```

Valeur de retour

Si l'appel réussit, le nombre d'octets effectivement extraits (> 0 et $\leq lg$), 0 si la socket a été fermée, -1 sinon (avec errno positionné).

Réception d'un message

```
ssize_t recv (int descripteur, // descripteur de socket  
              const void *msg, // * pointeur vers le premier octet où  
                               // sera stocké le message reçu *  
              size_t lg,       // le nombre max d'octets attendus  
              int flags)       // options de réception, 0 par défaut
```

Valeur de retour

Si l'appel réussit, le nombre d'octets effectivement extraits (> 0 et $\leq lg$), 0 si la socket a été fermée, -1 sinon (avec errno positionné).

Note

Cette fonction est bloquante. Dans quel cas ?

Réception d'un message - exemple

Prérequis

- Une socket correctement créée et connectée à une socket distante.
- *Espace mémoire alloué (statiquement ou dynamiquement) pour stocker le message reçu.*

Réception d'un message - exemple

Prérequis

- Une socket correctement créée et connectée à une socket distante.
- *Espace mémoire alloué (statiquement ou dynamiquement) pour stocker le message reçu.*

Que fait le code suivant ?

```
... // code incluant la création d'une socket dS et sa connexion  
int message [500];  
ssize_t res = recv(dS, message, sizeof(message), 0);
```

Réception d'un message - exemple

Prérequis

- Une socket correctement créée et connectée à une socket distante.
- *Espace mémoire alloué (statiquement ou dynamiquement) pour stocker le message reçu.*

Que fait le code suivant ?

```
... // code incluant la création d'une socket dS et sa connexion  
int message [500] ;  
ssize_t res = recv(dS, message, sizeof(message), 0) ;
```

Questions :

- Que signifie le commentaire "création d'une socket dS et sa connexion" coté client ?
- et coté serveur ?

Envoi d'un message

```
ssize_t send (int descripteur, // descripteur de socket  
              const void *msg,  /* pointeur vers le premier octet  
                                du message à envoyer */  
              size_t lg,        // le nombre d'octets du message  
              int flags)        // options d'envoi, 0 par défaut
```

Envoi d'un message

```
ssize_t send (int descripteur, // descripteur de socket
              const void *msg, /* pointeur vers le premier octet
                                du message à envoyer */
              size_t lg,       // le nombre d'octets du message
              int flags)       // options d'envoi, 0 par défaut
```

Valeur de retour

Si l'appel réussit, le nombre d'octets (> 0 et $\leq lg$) effectivement déposés dans le buffer associé à la socket, 0 si la socket a été fermée, -1 sinon (avec `errno` positionné).

Envoi d'un message

```
ssize_t send (int descripteur, // descripteur de socket
              const void *msg, /* pointeur vers le premier octet
                                du message à envoyer */
              size_t lg,       // le nombre d'octets du message
              int flags)       // options d'envoi, 0 par défaut
```

Valeur de retour

Si l'appel réussit, le nombre d'octets (> 0 et $\leq lg$) effectivement déposés dans le buffer associé à la socket, 0 si la socket a été fermée, -1 sinon (avec `errno` positionné).

Note

Cette fonction est bloquante. Dans quel cas ?

Envoi d'un message - exemple

Prérequis

- Une socket correctement créée et connectée **ou en attente de connexion**.
- Le message à envoyer correctement construit.

Envoi d'un message - exemple

Prérequis

- Une socket correctement créée et connectée **ou en attente de connexion**.
- Le message à envoyer correctement construit.

Que fait le code suivant ?

```
... // code incluant la création d'une socket dS et sa connexion
int message [500] ;
saisiValsTab(message, 500); // saisie des éléments du tableau
ssize_t res = send(dS, message, sizeof(message), 0);
```

Envoi d'un message - exemple

Prérequis

- Une socket correctement créée et connectée **ou en attente de connexion**.
- Le message à envoyer correctement construit.

Que fait le code suivant ?

```
... // code incluant la création d'une socket dS et sa connexion
int message [500] ;
saisiValsTab(message, 500) ; // saisie des éléments du tableau
ssize_t res = send(dS, message, sizeof(message), 0) ;
```

Question

Si la valeur de retour *res* est 1024, que s'est-il produit ? Que doit on faire pour que les 500 entiers soient envoyés ?

On reprend tout : exemple

Client (extrait)

```
dS= socket(PF_INET, SOCK_STREAM, 0);
struct sockaddr_in aS;
aS.sin_family = AF_INET;
inet_pton(AF_INET,argv[1],&(aS.sin_addr));
aS.sin_port = htons(atoi(argv[2]));
socklen_t lgA = sizeof(struct sockaddr_in);
connect(dS, (struct sockaddr *) &aS, lgA);
```

```
char * m = "Bonjour";
send(dS, m, 8, 0);
int r;
recv(dS, &r, sizeof(int), 0);
printf("reponse : %d", r);
close (dS);
```

Serveur (extrait)

```
dS= socket(PF_INET, SOCK_STREAM, 0);
struct sockaddr_in ad;
ad.sin_family = AF_INET;
ad.sin_addr.s_addr = INADDR_ANY;
ad.sin_port = htons(atoi(argv[1]));
bind(dS, (struct sockaddr*)&ad, sizeof(ad));
listen(dS, 7);
struct sockaddr_in aC;
socklen_t lg = sizeof(struct sockaddr_in);
dSC= accept(dS, (struct sockaddr*) &aC,&lg);
char msg [20];
recv(dSC, msg, sizeof(msg), 0);
printf("recu : %s", msg);
int r = 10;
send(dSC, &r, sizeof(int), 0);
close (dSC); close (dS);
```

On reprend tout : exemple

Client (extrait)

```
dS= socket(PF_INET, SOCK_STREAM, 0);
struct sockaddr_in aS;
aS.sin_family = AF_INET;
inet_pton(AF_INET, argv[1], &(aS.sin_addr));
aS.sin_port = htons(atoi(argv[2]));
socklen_t lgA = sizeof(struct sockaddr_in);
connect(dS, (struct sockaddr *) &aS, lgA);

char * m = "Bonjour";
send(dS, m, 8, 0);
int r;
recv(dS, &r, sizeof(int), 0);
printf("reponse : %d", r);
close (dS);
```

Serveur (extrait)

```
dS= socket(PF_INET, SOCK_STREAM, 0);
struct sockaddr_in ad;
ad.sin_family = AF_INET;
ad.sin_addr.s_addr = INADDR_ANY;
ad.sin_port = htons(atoi(argv[1]));
bind(dS, (struct sockaddr*)&ad, sizeof(ad));
listen(dS, 7);
struct sockaddr_in aC;
socklen_t lg = sizeof(struct sockaddr_in);
dSC= accept(dS, (struct sockaddr*) &aC, &lg);
char msg [20];
recv(dSC, msg, sizeof(msg), 0);
printf("recu : %s", msg);
int r = 10;
send(dSC, &r, sizeof(int), 0);
close (dSC); close (dS);
```

Démonstration...

Pour finir

- Un message en TCP est vu comme *un flux d'octets et peut être reçu et envoyé en une ou plusieurs fois*.
 - si un appel à *send(...)* ou *recv(...)* retourne un nombre d'octets inférieur au nombre attendu, il ne s'agit pas d'une erreur ! Ce cas est à prendre en compte à chaque utilisation de ces fonctions.
- Il n'est pas nécessaire d'être en réception pour recevoir un message.
- En TCP, il est aussi possible d'utiliser les fonctions *read(...)* et *write(...)* pour recevoir et envoyer des messages, ainsi que *sendto(...)*, *recvfrom(...)*.
- Les appels des fonctions : *connect(...)*, *accept(...)*, *send(...)*, *recv(...)*, *read(...)*, *write(...)*, *sendto(...)* et *recvfrom(...)* peuvent être *bloquants* .
 - Exercice : en considérant une application avec un client et un serveur et pour chaque fonction, donner un scénario aboutissant à une attente (blocage).

- 1 Introduction
- 2 Présentation des sockets
- 3 Communications en mode non connecté (UDP)
- 4 Communications en mode connecté (TCP)
- 5 Gestion de plusieurs clients (UDP et TCP)**

Introduction

- Les sections précédentes ont illustré l'utilisation des protocoles TCP et UDP dans le cas d'un seul client.
- Question actuelle : comment prendre en compte plusieurs clients ?
- Deux types de serveurs :
 - **itératif** : *traite un client à la fois et l'un après l'autre.*
 - **concurrent** : *traite plusieurs clients en parallèle (simultanément).*

Serveur itératif - UDP

Client 1



adr1

```
socket
sendto 'adrS'
recvfrom
...
close 'adr1'
```

Client 2



adr2

```
socket
while (...){
  sendto 'adrS'
  recvfrom
  ...
}
close 'adr2'
```

Serveur



adrS

```
socket
bind 'adrS'

recvfrom 'adrX'
...
sendto 'adrX'

close 'adrS'
```


Serveur itératif - UDP

Client 1



adr1

```
socket
sendto 'adrS'
recvfrom
...
close 'adr1'
```

Client 2



adr2

```
socket
while (...){
  sendto 'adrS'
  recvfrom
  ...
}
close 'adr2'
```

Serveur



adrS

```
socket
bind 'adrS'
```

```
recvfrom 'adrX'
```

```
...
sendto 'adrX'
```

```
close 'adrS'
```

Question

Comment faire pour prendre en compte les demandes des deux clients ?

Serveur itératif - UDP

Client 1



adr1

```
socket
sendto 'adrS'
recvfrom
...
close 'adr1'
```

Client 2



adr2

```
socket
while (...) {
  sendto 'adrS'
  recvfrom
  ...
}
close 'adr2'
```

Serveur



adrS

```
socket
bind 'adrS'
while(1){
  recvfrom 'adrX'
  ...
  sendto 'adrX'
}
close 'adrS'
```

Serveur itératif - UDP

Client 1



adr1

```
socket
sendto 'adrS'
recvfrom
...
close 'adr1'
```

Client 2



adr2

```
socket
while (...){
  sendto 'adrS'
  recvfrom
  ...
}
close 'adr2'
```

Serveur



adrS

```
socket
bind 'adrS'
while(1){
  recvfrom 'adrX'
  ...
  sendto 'adrX'
}
close 'adrS'
```

Note


Fermeture de la socket du serveur à l'arrêt du serveur.

Serveur itératif - TCP


Client 1


 **adr1**
 socket
 connect 'adrS'
 send
 recv
 ...
 close 'adr1'

Client 2

 **adr2**
 socket
 connect 'adrS'
 while (...){
 send
 recv
 ...
 }
 close 'adr2'

Serveur

 **adrS**
 socket
 bind 'adrS'
 listen 'adrS'

adrC 
 accept
 recv 'adrC'
 ...
 send 'adrC'
 close 'adrC'

 close 'adrS'

(Note: In the original image, 'adrC' and 'adrS' in the server code are underlined, and a curved arrow points from the 'recv' step to the 'send' step.)

Serveur itératif - TCP

Client 1



adr1

```
socket
connect 'adrS'
send
recv
...
close 'adr1'
```

Client 2



adr2

```
socket
connect 'adrS'
while (...){
  send
  recv
  ...
}
close 'adr2'
```

Serveur



adrS

```
socket
bind 'adrS'
listen 'adrS'
```



adrC

```
accept
recv 'adrC'
...
send 'adrC'
close 'adrC'
...
close 'adrS'
```

Question

Comment faire pour prendre en compte les demandes des deux clients ?

Serveur itératif - TCP

Client 1



adr1

```
socket
connect 'adrS'
send
recv
...
close 'adr1'
```

Client 2



adr2

```
socket
connect 'adrS'
while (...){
  send
  recv
  ...
}
close 'adr2'
```

Serveur



adrS

```
socket
bind 'adrS'
listen 'adrS'
while(1){
  accept
  recv 'adrC'
  ...
  send 'adrC'
  close 'adrC'
}
close 'adrS'
```

adrC




Serveur itératif - TCP


Client 1

 **adr1**
 socket
 connect 'adrS'
 send
 recv
 ...
 close 'adr1'

Client 2

 **adr2**
 socket
 connect 'adrS'
 while (...){
 send
 recv
 ...
 }
 close 'adr2'

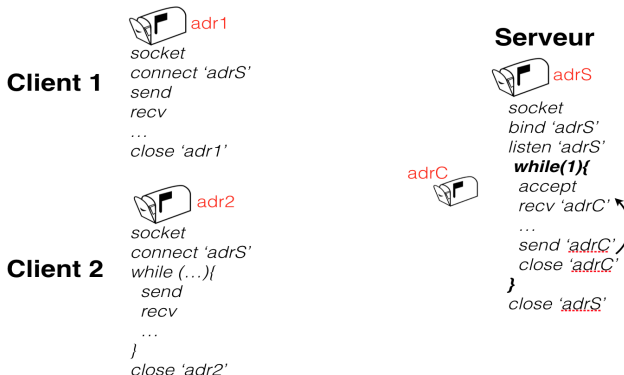
Serveur

 **adrS**
 socket
 bind 'adrS'
 listen 'adrS'
while(1){
 accept
 recv 'adrC'
 ...
 send 'adrC'
 close 'adrC'
}
 close 'adrS'

Note

Fermer la socket dédiée à un client à la fin du traitement de ce dernier.

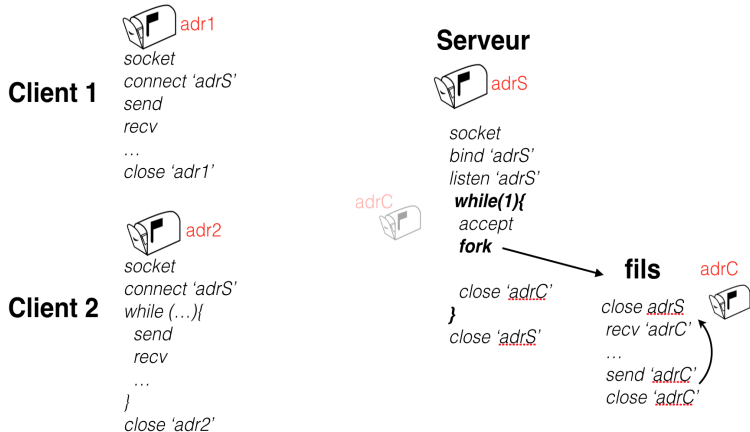
Serveur itératif - TCP - fin



Questions

Si le client 2 s'est connecté en premier et que le traitement de ses requêtes prend un temps considérable, que se passe-t-il pour le client 1 ? Comment résoudre ce problème ?

Serveur concurrent multiprocessus - TCP



Serveur concurrent multi-threads - TCP

Serveur concurrent multi-threads - TCP

Exercice

A faire en TP.

Serveur concurrent - UDP ?

Exercice

Peut on appliquer le même principe qu'en TCP ? Si oui, comment ?

Client 1



adr1

```
socket
sendto 'adrS'
recvfrom
...
close 'adr1'
```

Client 2



adr2

```
socket
while (...) {
  sendto 'adrS'
  recvfrom
  ...
}
close 'adr2'
```

Serveur



adrS

```
socket
bind 'adrS'
while(1){
  recvfrom 'adrX'
  ...
  sendto 'adrX'
}
close 'adrS'
```

Serveur - UDP, discussion

- Les messages reçus sur le serveur sont stockés dans le buffer associé à la socket et sont en attente de réception par l'application,
- *si le dialogue entre chaque client et le serveur se résume à des questions/réponses indépendantes et que le traitement des requêtes est court, un traitement itératif suffirait.*
- Si le dialogue est plus compliqué et que les questions/réponses ne sont pas indépendantes, on aura un mélange des demandes de plusieurs clients et leur traitement peut devenir difficile à gérer.
- Une solution est de *créer une nouvelle socket à la main pour maintenir un suivi par client (comme le fait la fonction accept)* ,
- l'adresse de cette nouvelle socket (en particulier le numéro de port) doit alors être envoyé au client après avoir reçu une requête de ce dernier.

Au final, quoi choisir entre (UDP ou TCP) et (itératif ou concurrent) ?

- La réponse est spécifique à chaque application.

Au final, quoi choisir entre (UDP ou TCP) et (itératif ou concurrent) ?

- La réponse est spécifique à chaque application.
- De manière générale :

Au final, quoi choisir entre (UDP ou TCP) et (itératif ou concurrent) ?

- La réponse est spécifique à chaque application.
- De manière générale :
 - Si les messages sont courts, les requêtes indépendantes et que leur traitement est court, pensez à un serveur UDP itératif.

Au final, quoi choisir entre (UDP ou TCP) et (itératif ou concurrent) ?

- La réponse est spécifique à chaque application.
- De manière générale :
 - Si les messages sont courts, les requêtes indépendantes et que leur traitement est court, pensez à un serveur UDP itératif.
 - Si les messages sont longs, les requêtes dépendantes ou que leur traitement est long, pensez à un serveur TCP concurrent.

Au final, quoi choisir entre (UDP ou TCP) et (itératif ou concurrent) ?

- La réponse est spécifique à chaque application.
- De manière générale :
 - Si les messages sont courts, les requêtes indépendantes et que leur traitement est court, pensez à un serveur UDP itératif.
 - Si les messages sont longs, les requêtes dépendantes ou que leur traitement est long, pensez à un serveur TCP concurrent.
 - Dans tous les cas, pensez aussi à d'autres critères, comme la performance et la fiabilité.

Au final, quoi choisir entre (UDP ou TCP) et (itératif ou concurrent) ?

- La réponse est spécifique à chaque application.
- De manière générale :
 - Si les messages sont courts, les requêtes indépendantes et que leur traitement est court, pensez à un serveur UDP itératif.
 - Si les messages sont longs, les requêtes dépendantes ou que leur traitement est long, pensez à un serveur TCP concurrent.
 - Dans tous les cas, pensez aussi à d'autres critères, comme la performance et la fiabilité.
- Et si les requêtes sont dépendantes et que leur traitement est court ?

Au final, quoi choisir entre (UDP ou TCP) et (itératif ou concurrent) ?

- La réponse est spécifique à chaque application.
- De manière générale :
 - Si les messages sont courts, les requêtes indépendantes et que leur traitement est court, pensez à un serveur UDP itératif.
 - Si les messages sont longs, les requêtes dépendantes ou que leur traitement est long, pensez à un serveur TCP concurrent.
 - Dans tous les cas, pensez aussi à d'autres critères, comme la performance et la fiabilité.
- Et si les requêtes sont dépendantes et que leur traitement est court ?
 - Un serveur concurrent peut être inadapté (mise en place des processus fils et commutations de contexte coûteux)

Au final, quoi choisir entre (UDP ou TCP) et (itératif ou concurrent) ?

- La réponse est spécifique à chaque application.
- De manière générale :
 - Si les messages sont courts, les requêtes indépendantes et que leur traitement est court, pensez à un serveur UDP itératif.
 - Si les messages sont longs, les requêtes dépendantes ou que leur traitement est long, pensez à un serveur TCP concurrent.
 - Dans tous les cas, pensez aussi à d'autres critères, comme la performance et la fiabilité.
- Et si les requêtes sont dépendantes et que leur traitement est court ?
 - Un serveur concurrent peut être inadapté (mise en place des processus fils et commutations de contexte coûteux)
- **Question** : peut-on gérer l'ensemble des clients simultanément dans un seul processus serveur (et un seul fil d'exécution) ?

Au final, quoi choisir entre (UDP ou TCP) et (itératif ou concurrent) ?

- La réponse est spécifique à chaque application.
- De manière générale :
 - Si les messages sont courts, les requêtes indépendantes et que leur traitement est court, pensez à un serveur UDP itératif.
 - Si les messages sont longs, les requêtes dépendantes ou que leur traitement est long, pensez à un serveur TCP concurrent.
 - Dans tous les cas, pensez aussi à d'autres critères, comme la performance et la fiabilité.
- Et si les requêtes sont dépendantes et que leur traitement est court ?
 - Un serveur concurrent peut être inadapté (mise en place des processus fils et commutations de contexte coûteux)
- **Question** : peut-on gérer l'ensemble des clients simultanément dans un seul processus serveur (et un seul fil d'exécution) ?
 - Oui, en utilisant le **multiplexage des entrées / sorties**

Pour finir

- Dans le cadre des TP, réfléchir à la nécessité d'avoir un serveur concurrent ou pas :
 - programmation multi-threads
 - il y a d'autres moyens de traiter plusieurs client "semi-simultanément" :
multiplexage des entrées / sorties (fonctions `select(...)`, `poll(...)`, ...).

Pour finir

- Dans le cadre des TP, réfléchir à la nécessité d'avoir un serveur concurrent ou pas :
 - programmation multi-threads
 - il y a d'autres moyens de traiter plusieurs client "semi-simultanément" : multiplexage des entrées / sorties (fonctions `select(...)`, `poll(...)`, ...).
- Toujours tester les valeurs de retour et quitter proprement vos programmes (libérations des espaces alloués dynamiquement, fermer les sockets, etc).

Pour finir

- Dans le cadre des TP, réfléchir à la nécessité d'avoir un serveur concurrent ou pas :
 - programmation multi-threads
 - il y a d'autres moyens de traiter plusieurs client "semi-simultanément" : multiplexage des entrées / sorties (fonctions `select(...)`, `poll(...)`, ...).
- Toujours tester les valeurs de retour et quitter proprement vos programmes (libérations des espaces alloués dynamiquement, fermer les sockets, etc).
- Favoriser le passage de paramètres à vos programmes (ou la saisie) et non le codage en dur des données supposées être modifiables.

Pour finir

- Dans le cadre des TP, réfléchir à la nécessité d'avoir un serveur concurrent ou pas :
 - programmation multi-threads
 - il y a d'autres moyens de traiter plusieurs client "semi-simultanément" : multiplexage des entrées / sorties (fonctions `select(...)`, `poll(...)`, ...).
- Toujours tester les valeurs de retour et quitter proprement vos programmes (libérations des espaces alloués dynamiquement, fermer les sockets, etc).
- Favoriser le passage de paramètres à vos programmes (ou la saisie) et non le codage en dur des données supposées être modifiables.
- Pour les TP, favoriser l'allocation d'un numéro de port par le système et utiliser les fonctions `getnameinfo(...)` pour connaître le numéro alloué.

Pour finir

- Dans le cadre des TP, réfléchir à la nécessité d'avoir un serveur concurrent ou pas :
 - programmation multi-threads
 - il y a d'autres moyens de traiter plusieurs client "semi-simultanément" : multiplexage des entrées / sorties (fonctions `select(...)`, `poll(...)`, ...).
- Toujours tester les valeurs de retour et quitter proprement vos programmes (libérations des espaces alloués dynamiquement, fermer les sockets, etc).
- Favoriser le passage de paramètres à vos programmes (ou la saisie) et non le codage en dur des données supposées être modifiables.
- Pour les TP, favoriser l'allocation d'un numéro de port par le système et utiliser les fonctions `getnameinfo(...)` pour connaître le numéro alloué.
- Tester vos applications en réseau (exécuter les clients et les serveurs sur des machines différentes).