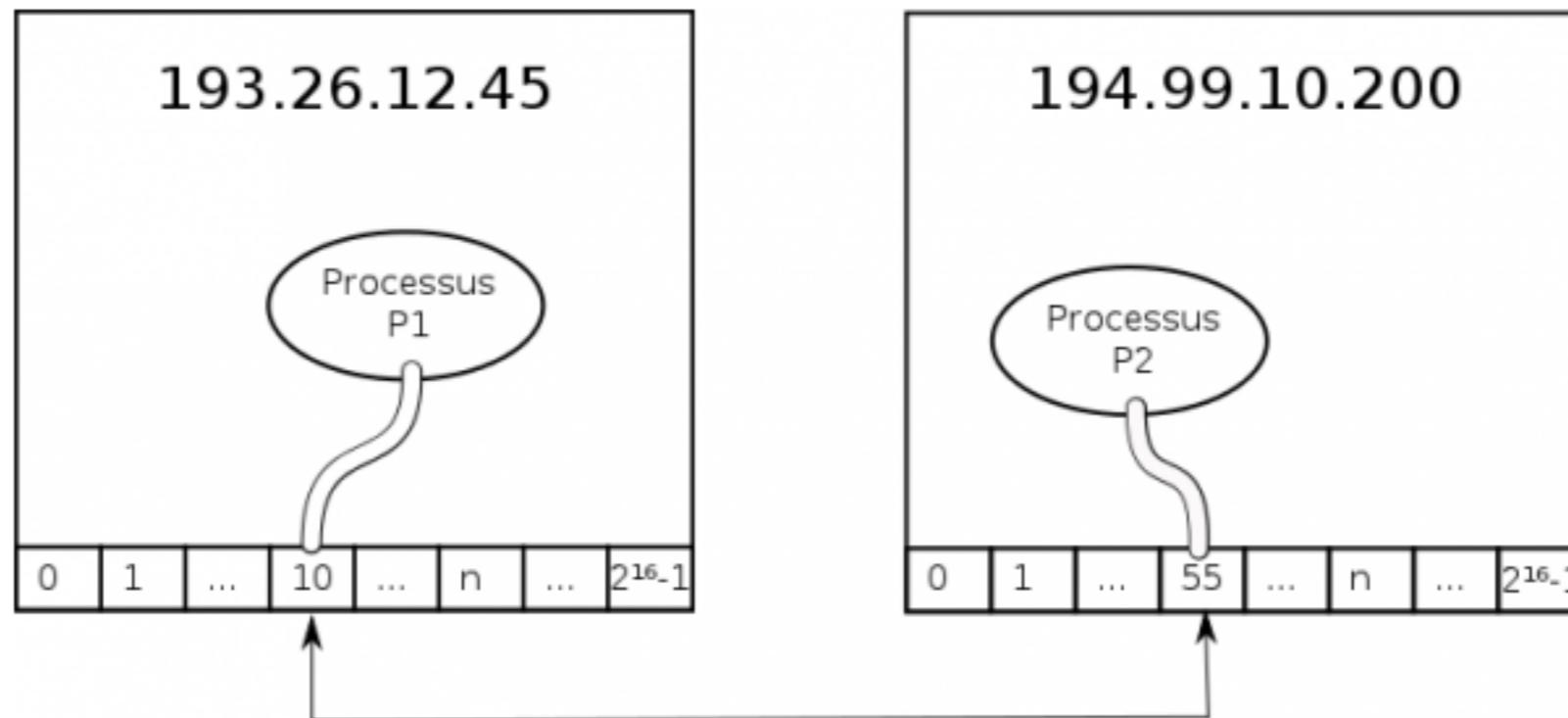


L'API des sockets en C

L'API des sockets en C

- Interface de programmation créée à l'université de Berkeley au début des années 1980 (Systèmes BSD).
- Accessible dans quasiment tous les langages (Java, C#, C++, Python, Perl, Ruby, ...)
- Une socket peut-être vue comme un tube “réseau” (mais pas seulement) bidirectionnel



L'API des sockets

Les algorithmes classiques d'un client et d'un serveur utilisant une socket UDP :

Client	Échanges	Serveur
1. Créer une socket UDP		1. Créer une socket UDP
2. Forger l'adresse du destinataire		2. Attacher la socket à un port
3. Préparer et émettre le message au destinataire	→	3. Attendre et lire une requête
4. Attendre et lire la réponse	← -	4. Préparer et émettre la réponse à l'émetteur de la requête
5. traiter la réponse		5. Boucler en 3.

Créer une socket UDP

La création d'une socket est réalisé par l'appel système : **socket(2)**

NOM

socket - Créer un point de communication

SYNOPSIS

```
#include <sys/types.h> /* Consultez NOTES */
#include <sys/socket.h>
```

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

...

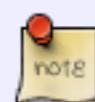
VALEUR RENVOYÉE

socket() renvoie un descripteur référençant la socket créée en cas de réussite. En cas d'échec -1 est renvoyé, et errno contient le code d'erreur.

Créer une socket UDP

Le paramètre **domaine** spécifie le domaine de communication souhaité (extrait) :

Nom	Utilisation	Page
AF_UNIX, AF_LOCAL	Communication locale	unix(7)
AF_INET	Protocoles Internet IPv4	ip(7)
AF_INET6	Protocoles Internet IPv6	ipv6(7)
AF_NETLINK	Interface utilisateur noyau	netlink(7)
AF_PACKET	Interface paquet bas-niveau	packet(7)



Dans le cadre de ce cours nous n'utiliserons que les familles AF_INET et AF_INET6

Créer une socket UDP

Le paramètre **type** spécifie la sémantique des communications (extrait) :

type	description
SOCK_STREAM	Support de dialogue garantissant l'intégrité, fournissant un flux de données binaires, et intégrant un mécanisme pour les transmissions de données hors-bande.
SOCK_DGRAM	Transmissions sans connexion, non garantie, de datagrammes de longueur maximale fixe.
SOCK_RAW	Accès direct aux données réseau.

- Certains types sont spécifiques à certaines familles.
- Le type associé à UDP est SOCK_DGRAM !

Créer une socket UDP

Le paramètre **protocole** spécifie l'entier associé au protocole (17 pour UDP, cf. /etc/protocoles ou getprotoent(3))

Lorsqu'il n'existe qu'un seul protocole associé à un couple (**domaine**, **type**) il est possible d'indiquer la valeur 0 pour le protocole.



C'est notamment le cas pour UDP dans (AF_INET, SOCK_DGRAM)

Créer une socket UDP

En résumé : créer une socket UDP dans le domaine IPv4 :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>

int main(void) {
    int s;
    /* ... */
    if((s=socket(AF_INET, SOCK_DGRAM, 0))==-1) {
        perror("socket");
        exit(1);
    }
    /* ... */
    return 0;
}
```

Client : forger une adresse IPv4

Avant de pouvoir émettre un datagramme UDP il nous faut définir l'adresse du destinataire en remplissant la structure **struct sockaddr_in** (cf. ip(7)).

```
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t      sin_family; /* famille d'adresses : AF_INET */
    in_port_t        sin_port;   /* port dans l'ordre des
                                octets réseau */
    struct in_addr sin_addr;   /* adresse Internet */
    unsigned char sin_zero[8]; /* doit être rempli de zéro */
};

/* Adresse Internet */
struct in_addr {
    uint32_t          s_addr;    /* adresse dans l'ordre des
                                octets réseau */
};
```

Client : forger une adresse IPv4

Le port (**sin_port**) et l'adresse (**sin_addr**) doivent respecter l'ordre des octets du réseau. Si données stockées dans variables (endianness de l'hôte) ⇒ conversion nécessaire.

<code>uint32_t htonl(uint32_t hostlong);</code>	long, ordre de l'hôte vers ordre réseau
<code>uint16_t htons(uint16_t hostshort);</code>	short, ordre de l'hôte vers ordre réseau

Conversion hôte (h) → réseau (n pour network) :

<code>uint32_t ntohl(uint32_t netlong);</code>	long, ordre réseau vers ordre de l'hôte
<code>uint16_t ntohs(uint16_t netshort);</code>	short, ordre réseau vers ordre de l'hôte

Client : forger une adresse IPv4

Construire adresse IPv4 à partir de son nom d'hôte :

```
#include <netdb.h>
extern int h_errno;
void perror(const char *s); // pour affichage erreur

struct hostent *gethostbyname(const char *name);
```

VALEUR RENVOYÉE

La fonction `gethostbyname()` renvoient un pointeur vers la structure `hostent`, ou un pointeur `NULL` si une erreur se produit, auquel cas `h_errno` contient le code d'erreur.



Obsolète mais présent dans de nombreux programmes, préférer `getnameinfo(3)`

Client : forger une adresse IPv4

La structure **struct hostent** dont l'adresse est retournée contient de nombreuses informations sur la machine dont le nom a été passé en argument à **gethostbyname**

```
struct hostent {
    char *h_name;          /* Nom officiel de l'hôte */
    char **h_aliases;      /* Liste d'alias */
    int   h_addrtype;       /* Type d'adresse de l'hôte */
    int   h_length;         /* Longueur de l'adresse */
    char **h_addr_list;    /* Liste d'adresses (ordre réseau) */
}

#define h_addr h_addr_list[0] /* for backward compatibility */
```

Client : forger une adresse IPv4



Plusieurs adresses IP (`char **h_addr_list`) peuvent être associées à une machine. Nous utiliserons la première du tableau.

Client : forger une adresse IPv4

Code d'exemple (le nom de la machine destinataire est supposé être passé en premier argument, le port d'écoute du destinataire en second argument).

```
#include <netdb.h>          /* pour gethostbyname*/
#include <string.h>          /* pour memset */
/* ... */
struct sockaddr_in addr;
struct hostent * hostent;
/* ... */
if ((hostent=gethostbyname(argv[1]))==NULL) {
    perror("gethostbyname");
    exit(1);
}

addr.sin_family = AF_INET;
addr.sin_port = htons(atoi(argv[2]));
addr.sin_addr = *((struct in_addr *)hostent->h_addr);
memset(addr.sin_zero, '\0', sizeof addr.sin_zero);
/* ... */
```

Client : émettre un datagramme UDP

Appel système **sendto(2)** :

SYNOPSIS

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

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

DESCRIPTION

L'appel système `sendto()` permet de transmettre un message à destination d'une autre socket.

VALEUR RENVOYÉE

En cas d'envoi réussi, `sendto` renvoie le nombre de caractères envoyés. En cas d'erreur, -1 est renvoyé, et `errno` contient le code d'erreur.

Client : émettre un datagramme UDP

- Détails des arguments :

- **int sockfd** : le descripteur de fichier de la socket émettrice
- **const void *buf** : le message à envoyer
- **size_t len** : nombre d'octets du message
- **int flags** : options éventuelles. 0 si pas d'options.
- **const struct sockaddr *dst_addr** : adresse du destinataire.
- **socklen_t addrlen** : longueur de l'adresse du destinataire.

Le pointeur **struct sockaddr *** utilisé dans le prototype nécessite un transtypage du type **struct sockaddr_in ***:



```
(struct sockaddr *) &dst_addr
```

Client : émettre un datagramme UDP

Code d'exemple d'émission de la chaîne de caractères “Hello” :

```
char msg[]="Hello";
/* ... */
if(sendto(s, msg, strlen(msg), 0, (struct sockaddr*) &dst_addr,
  sizeof(dst_addr))== -1) {
    perror("sendto"); exit(1);
}
```

Client : recevoir un datagramme UDP

Appel système `recvfrom(2)` :

SYNOPSIS

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

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

DESCRIPTION

L'appel système `recvfrom()` est utilisé pour recevoir des messages depuis une socket.

VALEUR RENVOYÉE

`recvfrom()` renvoie le nombre d'octets reçus si elle réussit, ou -1 si elle échoue.

Client : recevoir un datagramme UDP

- Détails des arguments :

- **int sockfd** : le descripteur de fichier de la socket sur laquelle on souhaite lire
- **void *buf** : emplacement où sera placé le message lu
- **size_t len** : nombre d'octets réservés pour le message (**recvfrom** n'écrira jamais plus de len octets à l'emplacement buf).
- **int flags** : options éventuelles. 0 si pas d'options.
- **const struct sockaddr *src_addr** : emplacement mémoire où sera placé l'adresse de l'émetteur (très utile pour répondre).
- **socklen_t *addrlen** : adresse de l'entier ou sera placée la longueur de l'adresse de l'émetteur.

Client : recevoir un datagramme UDP

Code d'exemple de réception d'un datagramme contenant une chaîne de caractères :

```
/* ... */
#define DGRAM_MAX 1024
/* ... */
char response[DGRAM_MAX];

struct sockaddr_in src_addr;
socklen_t len_src_addr;
int ret;
/* ... */
len_src_addr=sizeof(src_addr);
if((ret=recvfrom(s, response, DGRAM_MAX-1, 0, \
    (struct sockaddr*)&src_addr, &len_src_addr))==-1) {
    perror("recvfrom"); exit(1);
}
response[ret]='\0';
/* ... */
```

Client : code complet

Ci-dessous le code complet du client :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>

#include <netdb.h>          /* pour gethostbyname */
#include <string.h>          /* pour memset */

#define DGRAM_MAX 1024 /* taille MAX en réception */

void usage() {
    fprintf(stderr, "usage : client_ipv4 hostname port\n");
    exit(1);
}
```

Client : code complet

```
int main(int argc, char **argv) {
    struct sockaddr_in dst_addr, src_addr;
    struct hostent * hostent;
    int s, ret;
    socklen_t len_src_addr;

    char response[DGRAM_MAX];

    char msg[]{"Hello"};

    /* Vérification des arguments */
    if(argc!=3) {
        fprintf(stderr,"Erreur : Nb args !\n");
        usage();
    }
```

Client : code complet

```
/* Cr ation de la socket IPv4 */
if((s=socket(AF_INET, SOCK_DGRAM, 0))==-1) {
    perror("socket"); exit(1);
}

/* R cup ration de l'adresse IPv4 du destinataire */
if((hostent=gethostbyname(argv[1]))==NULL) {
    herror("gethostbyname"); exit(1);
}

/* Remplissage de l'adresse du destinataire */
dst_addr.sin_family = AF_INET;
dst_addr.sin_port = htons(atoi(argv[2]));
dst_addr.sin_addr = *((struct in_addr *)hostent->h_addr);
memset(dst_addr.sin_zero, '\0', sizeof(dst_addr.sin_zero));
```

Client : code complet

```
/* Émission du datagramme */
if(sendto(s, msg, strlen(msg)+1, 0,
          (struct sockaddr*) &dst_addr, sizeof(dst_addr)) == -1) {
    perror("sendto"); exit(1);
}

/* Attente et lecture de la réponse */
len_src_addr = sizeof(src_addr);
if((ret=recvfrom(s, response, DGRAM_MAX-1, 0,
                 (struct sockaddr*) &src_addr, &len_src_addr)) == -1) {
    perror("recvfrom"); exit(1);
}
response[ret] = 0;

/* Traitement de la réponse */
puts(response);

return 0;
}
```

Serveur : Généralités

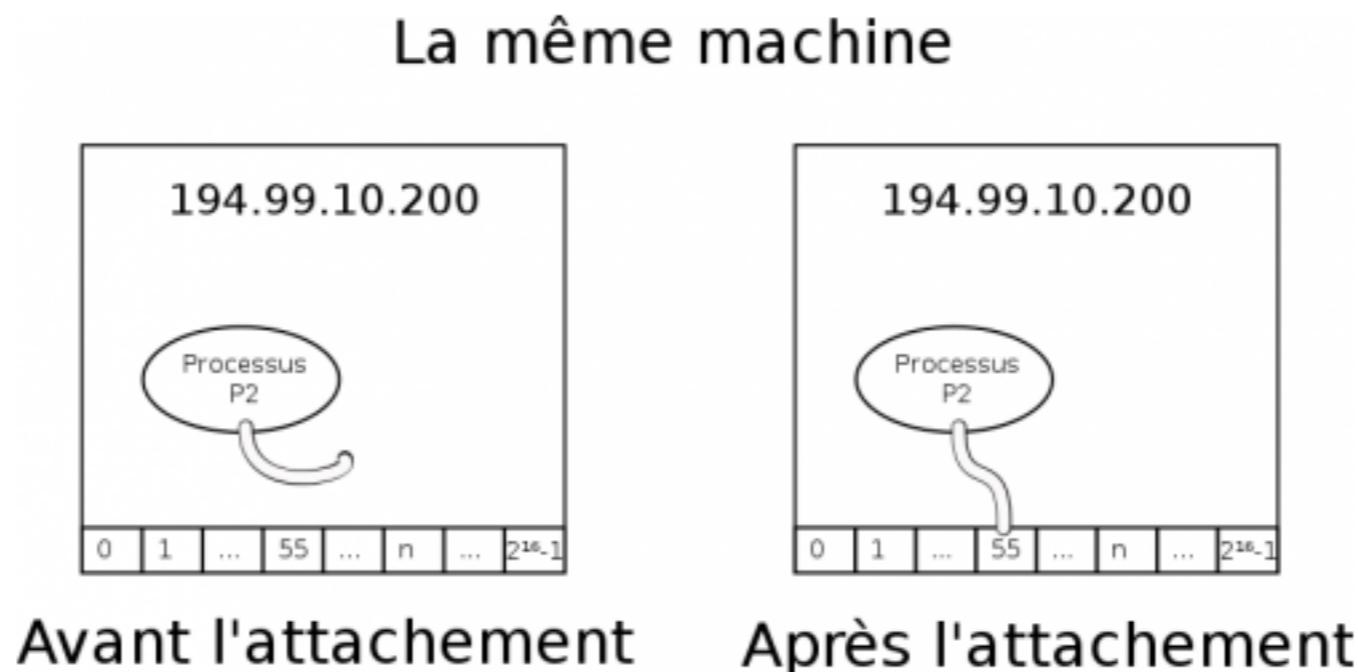
Beaucoup de séquences de l'algorithme du serveur (rappelées ci-dessous) ont déjà étudiées puisqu'identiques chez le client.

Client	Échanges	Serveur
1. Créer une socket UDP		1. Créer une socket UDP
2. Forger l'adresse du destinataire		2. Attacher la socket à un port
3. Préparer et émettre le message au destinataire	→	3. Attendre et lire une requête
4. Attendre et lire la réponse	← -	4. Préparer et émettre la réponse à l'émetteur de la requête
5. traiter la réponse		5. Boucler en 3.

C'est notamment le cas de la création de la socket, de la réception et de l'émission d'un datagramme.

Serveur : Attachement de la socket

Juste après sa création, la socket n'est pas encore attachée à un port.



Note : L'appel système :

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

permet d'obtenir des informations sur la socket (adresse IP et port d'attachement).

Serveur : Attachement de la socket

Dans le cas du client l'attachement à un port a eu lieu

- automatiquement lors de la première émission.
- le port d'attachement choisi par le système est un port libre quelconque (>1023)

Dans le cas d'un serveur le port doit généralement :

- est fixé explicitement par le programme puisque c'est par ce numéro que les clients identifient le service.

Serveur : Attachement de la socket

L'attachement est réalisé par l'appel système **bind(2)** :

NOM

bind - Fournir un nom à une socket

SYNOPSIS

```
#include <sys/types.h>          /* Consultez NOTES */  
#include <sys/socket.h>
```

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

DESCRIPTION

Dans le cas d'IPv4, attache la socket à l'adresse et au port indiqués dans `addr` (de type réel `sockaddr_in`). L'argument `addrlen` est la taille de la structure pointée par `addr`.

Serveur : Attachement de la socket

Rôle du champ `sin_addr` pour l'adresse d'attachement :

- si fixé à une adresse d'une des interfaces de la machine, la socket n'écouterait que les datagrammes en provenance de cette interface.
- si fixé à la valeur `INADDR_ANY` la socket écouterait les messages en provenance de toutes les interfaces de la machine.

Serveur : Attachement de la socket

Ci-dessous le code classique d'attachement d'une socket par un serveur (adresse spéciale INADDR_ANY, port défini dans argv[1]).

```
/* Attachement de la socket */
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(atoi(argv[1]));
memset(addr.sin_zero, '\0', sizeof(addr.sin_zero));

if(bind(s, (struct sockaddr *)&addr, sizeof addr)) {
    perror("bind");
    exit(1);
}
```

Serveur : Émission d'une réponse

- A la lecture d'une requête par `recvfrom` la structure `sockaddr_in` passée en argument est remplie avec l'adresse de l'émetteur.
- Cette structure pourra directement être utilisée dans le `sendto` pour envoyée la réponse.

Serveur : code complet

Ci-dessous le code d'un serveur répondant aux requêtes des clients (chaîne de caractères) en leur retournant leur requête en MAJUSCULES :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>
#include <netdb.h>      /* pour gethostbyname */
#include <string.h>      /* pour memset */
#include <ctype.h>      /* pour toupper */

#define DGRAM_MAX 1024 /* taille MAX en réception */

void usage() {
    fprintf(stderr, "usage : serveur_ipv4 port\n");
    exit(1);
}
```

Serveur : code complet

```
int main(int argc, char **argv) {
    struct sockaddr_in addr, src_addr;
    int s, ret;
    socklen_t len_src_addr;

    char request[DGRAM_MAX];

    /* Vérification des arguments */
    if(argc!=2) {
        fprintf(stderr,"Erreur : Nb args !\n");
        usage();
    }
```

Serveur : code complet

```
/* Cr ation de la socket IPv4 */
if((s=socket(AF_INET, SOCK_DGRAM, 0))==-1) {
    perror("socket"); exit(1);
}

/* Attachement de la socket */
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl (INADDR_ANY);
addr.sin_port = htons(atoi(argv[1]));
memset(addr.sin_zero, '\0', sizeof(addr.sin_zero));

if(bind(s, (struct sockaddr *)&addr, sizeof addr)) {
    perror("bind"); exit(1);
}
```

Serveur : code complet

```
while(1) {  
  
    /* Attente et lecture d'une requête */  
    len_src_addr=sizeof src_addr;  
    if((ret=recvfrom(s, request, DGRAM_MAX-1, 0,  
                    (struct sockaddr* ) &src_addr, &len_src_addr))==-1) {  
        perror("recvfrom"); exit(1);  
    }  
    request[ret]=0;  
  
    /* traitement de la requête(passage en majuscule) */  
    {  
        int i=0;  
  
        while(request[i]) {  
            request[i]=toupper(request[i]);  
            ++i;  
        }  
    }  
}
```

Serveur : code complet

```
/* Émission du datagramme réponse */

if(sendto(s, request, strlen(request)+1, 0, \
          (struct sockaddr*)&src_addr, sizeof(src_addr))== -1) {
    perror("sendto");
    exit(1);
}

return 0;
}
```

Diffusion

Diffuser une trame UDP à l'adresse de broadcast nécessite l'activation de cette capacité pour la socket UDP :

```
/* Activer la possibilité de diffuser (broadcast) */
{
    int on=1;
    setsockopt (sock_fd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
}
```

L'argument `sock_fd` est le descripteur de la socket.

Pour plus d'information sur l'appel système **setsockopt(2)** consultez sa page de manuel ainsi que celle d'`udp(7)`.

Une fois l'activation de la capacité de diffusion faite sur la socket vous pourrez transmettre en utilisant l'adresse de diffusion.

Autres fonctions souvent utiles

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

Convertit IPv4 sous forme de chaîne de caractères en son adresse numérique (ordre des octets du réseau).

```
char *inet_ntoa(struct in_addr in);
```

Convertit une adresse numérique (ordre des octets du réseau) en sa représentation sous forme de chaîne de caractères.

```
struct hostent *gethostbyaddr(const void *addr,  
    socklen_t len, int type);
```

Renvoie vers une structure struct hostent associée à l'adresse IP au format numérique (par exemple le champ sin_addr d'une structure struct sockaddr_in).

Débogage

Pour lister les sockets UDP attachées dans le système :

```
netstat -aup
```

Pour tracer les appels systèmes réalisés par un programme :

```
strace ./client localhost 3000
```

Pour émettre une trame UDP en ligne de commande

```
echo "hello" | netcat -u localhost 3000
```

Débogage

Pour “sniffer” les trames UDP sur une interface (nécessite d'être root)

```
tcpdump -i eth0 -X -vvv udp
```

L'outil graphique d'analyse du traffic réseau wireshark (nécessite d'être root)

```
wireshark
```

IPv6 - adresse IPv4 mappée IPv6

De nombreuses techniques sont utilisées pour faire cohabiter IPv4 et IPv6. L'une d'elles, définie par la [RFC 3493](#), introduit la notion d'adresses IPv4 mappée IPv6 et sont de la forme :

::ffff:0:0/96	Exemple : ::ffff.195.1.2.3
---------------	----------------------------

Ces adresses n'apparaissent jamais sur le réseau mais sont utilisées en interne dans la pile IPv6 des systèmes d'exploitation pour dialoguer avec des correspondants s'adressant à eux avec une adresse IPv4 .

IPv6 - Actualité du déploiement

- Les systèmes d'exploitation modernes incluent les sockets IPv6.
- 4% des requêtes accès à google en IPv6
<http://www.google.com/intl/en/ipv6/statistics.html>
 - Mais croissance exponentielle
- Cohabitation IPv4/v6 sans doute de longue durée
 - Possible car Dual Stack

Socket IPv6

- L'API des sockets a été adaptée en conséquence et propose :
 - Des nouveaux types et constantes adaptés à IPv6
 - De nouvelles fonctions utilitaires
- Les nouvelles fonctions proposées rendent certaines fonctions spécifiques à IPv4 obsolètes :
 - notamment **gethostbyname(3)/gethostbyaddr(3)**

Socket IPv6

Comment faire des programmes compatibles à la fois IPv4 et IPv6 ?

Première solution :

- Utiliser la capacité proposée par les systèmes d'exploitation modernes contenant à la fois la couche IPv4 et IPv6 d'autoriser une socket IPv6 à gérer un interlocuteur IPv4 en utilisant son adresse IPv6 “mappée” automatiquement.

Socket IPv6 : IPv4 mappé sur IPv6

Cette solution, simple sur le papier, présente des inconvénients dont voici quelques uns :



- Une socket IPv4 et IPv6 n'acceptent pas les mêmes options.
- Une adresse mappée peut-être utilisée pour contourner certaines règles de pare-feux

Certains systèmes d'exploitation interdisent par défaut cette capacité.

Socket IPv6

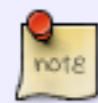
Seconde solution :

- Gérer 2 sockets : Une IPv4 et une IPv6.
- Cette solution complexifie un peu le code car elle nécessite de “surveiller” (cf. `select(2)` pour info) sur laquelle des 2 sockets la communication est initiée.
- Bien que très intéressante cette seconde technique sort du cadre de ce cours.

Socket IPv6

Pour notre cours: première technique malgré ses défauts

Il est possible d'interdire à une socket IPv6 de prendre en compte les adresses IPv4 mappées via l'option **IPV6_V6ONLY** placée sur la socket (cf. **getsockopt(2)** et **ipv6(7)**)



Les méthodes présentées ci-après sont très généralistes puisqu'elles peuvent s'appliquer quelque soit la version IP !

Passage de Socket IPv4 à IPv6

Pour vous présenter les “nouveautés” de l'API des sockets introduites pour la prise en compte d'IPv6 nous allons porter le code typé IPv4 vu précédemment.

Les parties de code typé IPv4 étaient les suivantes :

- **struct sockaddr_in addr, src_addr;**
- **s=socket(AF_INET, SOCK_DGRAM, 0)**
- **hostent=gethostbyname(argv[1])**
- **dst_addr.sin_family = AF_INET;**
- **addr.sin_addr.s_addr = htonl (INADDR_ANY);**
- **memset(addr.sin_zero, '\0', sizeof(addr.sin_zero));**

Socket IPv6

- nouvelles constantes :
 - AF_INET6 : famille IPv6
- Nouvelles structures :

```
struct sockaddr_in6 {
    sa_family_t      sin6_family;    /* AF_INET6 */
    in_port_t        sin6_port;     /* numéro de port */
    uint32_t         sin6_flowinfo; /* information de flux IPv6 */
    struct in6_addr sin6_addr;     /* adresse IPv6 */
    uint32_t         sin6_scope_id; /* Scope ID (nouveauté 2.4) */
};

struct in6_addr {
    unsigned char   s6_addr[16];    /* adresse IPv6 */
};
```

Socket IPv6

La structure **struct sockaddr_in6** est plus grande que la structure **struct sockaddr_in**.

Il est nécessaire d'utiliser la nouvelle structure générique **struct sockaddr_storage** de taille suffisante pour contenir soit une adresse IPv4 (**struct sockaddr_in**) soit une adresse IPv6 (**struct sockaddr_in6**)

Socket IPv6

Exemple de réception d'une message en provenance d'une socket IPv4 ou d'une socket IPv6 :

```
struct sockaddr_storage src_addr;
socklen_t len_src_addr = sizeof src_addr;
/* ... */
if((ret=recvfrom(s, request, DGRAM_MAX, 0,
    (struct sockaddr*)&src_addr, &len_src_addr))==-1) {
    perror("recvfrom"); exit(1);
}
```

getaddrinfo(3)

La fonction **getaddrinfo(3)**, présentée ci-après, rend obsolète la fonction **gethostbyname(3)** déjà étudiée.

- **getaddrinfo(3)** est compatible IPv4 et IPv6
- Comme **gethostbyname(3)** elle effectuera la résolution de l'IP à partir d'un nom d'hôte
- Elle remplira des structures **struct sock_addrin** et **struct sock_addrin6** prêtes à l'emploi pour :
 - l'émission via **sendto(2)**
 - la connexion via **connect(2)** (vu après lors de l'étude de TCP).
 - l'attachement via **bind(2)**

getaddrinfo(3)

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

DESCRIPTION

Étant donnés `node` et `service`, qui identifient un hôte Internet et un service, `getaddrinfo()` renvoie une ou plusieurs structure `addrinfo`, chacune d'entre elles contenant une adresse Internet...

VALEUR RENVOYÉE

`getaddrinfo()` renvoie 0 si elle réussit, ou un code d'erreur non nul qui pourra être passé à `'gai_strerror(3)'` pour obtenir la chaîne contenant une traduction compréhensible.

getaddrinfo(3)

Ses arguments sont :

- **const char *node** : nom d'hôte (ex : “opale.u-clermont1.fr”, “bigboss”, “192.168.1.2”).
- **const char *service** : nom du service sous la forme d'un numéro de port ou d'un nom tel que défini dans **/etc/services**.
- **const struct addrinfo *hints** : permet de spécifier les critères de sélection des adresses et sera étudié juste après la présentation de la structure **struct addrinfo**
- **struct addrinfo **res** : liste chaînée qui contiendra après l'appel à la fonction la liste des adresses répondant aux critères.

getaddrinfo(3)



Il sera nécessaire de libérer l'espace mémoire alloué dynamiquement pour la liste chaînée via un appel à `void freeaddrinfo(struct addrinfo *res);`

getaddrinfo(3)

La structure **struct addinfo** sert à la fois :

- à spécifier les critères de sélection (argument **hint**)
- de conteneur pour les adresses trouvées (argument **res**).

```
struct addrinfo {  
    int          ai_flags;      /* options supplémentaires */  
    int          ai_family;     /* famille (AF_INET, ...) */  
    int          ai_socktype;   /* type (SOCK_DGRAM, ...) */  
    int          ai_protocol;   /* protocole (0 pour auto) */  
    socklen_t    ai_addrlen;    /* longueur de ai_addr */  
    struct sockaddr *ai_addr;   /* adresse */  
    char         *ai_canonname; /* nom officiel (cf. note) */  
    struct addrinfo *ai_next;   /* adresse suivante (ou NULL) */  
};
```

getaddrinfo(3)



ai_canonname est initialisé (dans la première adresse de la liste) ssi l'attribut **AI_CANONNAME** est présent dans **ai_flags**.

getaddrinfo(3)

Exemple n°1 : côté client UDP pour construire l'adresse destination.

```
struct addrinfo hints, *result;
int ret;
/* ... */
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_flags = 0;
hints.ai_family = AF_UNSPEC;      /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_DGRAM;   /* Datagram socket */
hints.ai_protocol = 0;            /* Any protocol */
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;

ret = getaddrinfo(argv[1], argv[2], &hints, &result);
if (ret != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
    exit(EXIT_FAILURE);
}
/* ... */
```

getaddrinfo(3)

Remarques :

- La création de la socket (**socket(2)**) pourra être réalisée avec les champs de la première structure **struct addrinfo** du résultat (**result**).
- L'adresse du destinataire et sa taille sont placées dans les champs **ai_addr** et **ai_addrlen** prêt à être utilisés par **sendto(2)**.

getaddrinfo(3)

```
/* ... */
/* Cr ation de la socket IPv4/IPv6 */
if((s=socket(result->ai_family, result->ai_socktype,
    result->ai_protocol))==-1) {
    perror("socket"); exit(1);
}

/*  mission du datagramme */

if(sendto(s, msg, strlen(msg)+1, 0,
    result->ai_addr, result->ai_addrlen)==-1) {
    perror("sendto"); exit(1);
}

/* ... */
freeaddrinfo(result); /* d s que result ne sert plus */
/* ... */
```

Client : code complet

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>

#include <netdb.h>          /* pour getaddrinfo*/
#include <string.h>          /* pour memset */

#include <arpa/inet.h>      /* pour inet_ntop */

#define DGRAM_MAX 1024 /* taille MAX en réception */

void usage() {
    fprintf(stderr, "usage : client hostname port\n");
    exit(1);
}
```

Client : code complet

```
int main(int argc, char **argv) {
    struct sockaddr_storage src_addr;
    int s, ret;
    socklen_t len_src_addr;

    struct addrinfo hints, *result;

    char response[DGRAM_MAX];

    char msg[]="Hello";

    /* Vérification des arguments */
    if(argc!=3) {
        fprintf(stderr,"Erreur : Nb args !\n");
        usage();
    }
```

Client : code complet

```
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_flags = 0;
hints.ai_family = AF_UNSPEC;           /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_DGRAM;        /* Datagram socket */
hints.ai_protocol = 0;                 /* Any protocol */
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;

ret = getaddrinfo(argv[1], argv[2], &hints, &result);
if (ret != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
    exit(EXIT_FAILURE);
}
```

Client : code complet

```
/* Cr ation de la socket IPv4/IPv6 */
if((s=socket(result->ai_family, result->ai_socktype,
    result->ai_protocol))==-1) {
    perror("socket"); exit(1);
}

/*  mission du datagramme */
if(sendto(s, msg, strlen(msg)+1, 0, \
    result->ai_addr, result->ai_addrlen)==-1) {
    perror("sendto"); exit(1);
}

freeaddrinfo(result); // si result ne sert plus
```

Client : code complet

```
/* Attente et lecture de la réponse */
len_src_addr=sizeof(src_addr);
if((ret=recvfrom(s, response, DGRAM_MAX-1, 0,
    (struct sockaddr*)&src_addr, &len_src_addr))==-1) {
    perror("recvfrom"); exit(1);
}
response[ret]=0;

/* Traitement de la réponse */
puts(response);

return 0;
}
```

getaddrinfo(3)

Exemple n°2 : côté serveur UDP pour construire l'adresse d'attachement.

```
struct addrinfo hints, *result;
int ret;
/* ... */
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_flags = AI_PASSIVE;      /* For wildcard IP address */
hints.ai_family = AF_INET6;       /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_DGRAM;   /* Datagram socket */
hints.ai_protocol = 0;            /* Any protocol */
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;

ret = getaddrinfo(NULL, argv[1], &hints, &result);
if (ret != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
    exit(EXIT_FAILURE);
}
/* ... */
```

getaddrinfo(3)

Remarques :

- Le flag **AI_PASSIVE** est précisé pour indiquer que nous souhaitons obtenir l'adresse “jocker” (wildcard address : **INADDR_ANY** pour les adresses IPv4, **IN6ADDR_ANY_INIT** pour les adresses IPv6).
- l'hôte ici est fixé à **NULL** (premier argument de **getaddrinfo**) pour la même raison (attachement sur n'importe quelle interface).
- La création de la socket (**socket(2)**) et son attachement (**bind(2)**) pourront être réalisés avec les champs de la première structure **struct sockaddr** du résultat (**result**).

getaddrinfo(3)

```
/* ... */
if((s=socket(result->ai_family, result->ai_socktype,
    result->ai_protocol))==-1) {
    perror("socket"); exit(1);
}

if (bind(s, result->ai_addr, result->ai_addrlen) == -1) {
    perror("bind"); exit(1);
}

freeaddrinfo(result);
/* ... */
```

Serveur : code complet

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>

#include <netdb.h>
#include <string.h>      /* pour memset */
#include <ctype.h>        /* pour toupper */
#include <arpa/inet.h>   /* pour inet_ntop */

#define DGRAM_MAX 1024 /* taille MAX en réception */

void usage() {
    fprintf(stderr, "usage : serveur port\n");
    exit(1);
}
```

Serveur : code complet

```
int main(int argc, char **argv) {
    int s, ret;
    socklen_t len_src_addr;

    struct addrinfo hints, *result;

    struct sockaddr_storage src_addr;

    char request[DGRAM_MAX];

    /* Vérification des arguments */
    if(argc!=2) {
        fprintf(stderr,"Erreur : Nb args !\n");
        usage();
    }
```

Serveur : code complet

```
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_flags = AI_PASSIVE; /* For wildcard IP address */
hints.ai_family = AF_INET6; /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */
hints.ai_protocol = 0; /* Any protocol */
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;

ret = getaddrinfo(NULL, argv[1], &hints, &result);
if (ret != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
    exit(EXIT_FAILURE);
}
```

Serveur : code complet

```
/* Cr ation de la socket IPv4/IPv6 */
if((s=socket(result->ai_family, result->ai_socktype,
    result->ai_protocol))==-1) {
    perror("socket"); exit(1);
}

/* Attachement de la socket */

if (bind(s, result->ai_addr, result->ai_addrlen) == -1) {
    perror("bind"); exit(1);
}

freeaddrinfo(result);
```

Serveur : code complet

```
while(1) {  
  
    /* Attente et lecture d'une requête */  
    len_src_addr=sizeof src_addr;  
    if((ret=recvfrom(s, request, DGRAM_MAX-1, 0,  
                    (struct sockaddr*) &src_addr, &len_src_addr))==-1) {  
        perror("recvfrom"); exit(1);  
    }  
    request[ret]=0;  
  
    puts(request);  
  
    /* traitement de la requête(passage en majuscule) */  
    {  
        int i=0;  
  
        while(request[i]) {  
            request[i]=toupper(request[i]);  
            ++i;  
        }  
    }  
}
```

Serveur : code complet

```
    /* Émission du datagramme réponse */

    if(sendto(s, request, strlen(request)+1, 0,
              (struct sockaddr*)&src_addr, sizeof(src_addr)) == -1) {
        perror("sendto");
        exit(1);
    }

    return 0;
}
```

Autres fonctions utiles

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,  
    char *host, size_t hostlen,  
    char *serv, size_t servlen, int flags);
```

DESCRIPTION

La fonction `getnameinfo()` est la réciproque de `getaddrinfo(3)` : elle convertit une adresse de socket en un hôte et un service correspondants, de façon indépendante du protocole...

```
const char *inet_ntop(int af, const void *src,  
    char *dst, socklen_t size);
```

DESCRIPTION

Cette fonction convertit une structure d'adresse réseau `src` de la famille d'adresse `af` en une chaîne de caractères...

Socket TCP

Les algorithmes classiques d'un client et d'un serveur communiquant par TCP :

Client	Échanges	Serveur
1. Créer une socket TCP		1. sock_écoute =Créer une socket TCP
2. Forger l'adresse du serveur		2. Attacher la socket à un port
		3. Définir la taille de la liste des connexions pendantes
3. Se connecter au serveur	→	4. sock_service = Attendre une connexion (d'un client)
4. Échanger des informations	↔	5. Échanger des informations via sock_service
5. Terminer la connexion	↔	6. Terminer la connexion et boucler en 4.

Socket TCP

Outre le type de la socket (TCP) ce qui change par rapport aux sockets UDP :

- Chez le client :
 - La notion de connexion (**connect(2)**)
 - La notion de terminaison de connexion (**close(2)/shutdown(2)**)
- Du côté du serveur :
 - La notion de taille de liste de connexions pendantes (**listen(2)**).
 - représente nombre de clients maximum mis en attente
 - La notion d'attente de connexion (**accept(2)**), de socket d'écoute et de socket de service
 - socket d'écoute réservée aux demandes de connexions
 - socket de service obtenue lors de l'acceptation d'une connexion et donc connectée à un unique client
 - Et également la notion de terminaison.

Socket TCP

- Il existe des appels systèmes dédiés (**send(2)/recv(2)**).
- L'appel systèmes **socket(2)** et la fonction **getaddrinfo(3)** ne nécessiteront qu'une légère adaptation :
 - remplacer SOCK_DGRAM (UDP) par SOCK_STREAM (TCP).
- L'appel système **bind(2)** quant-à lui s'utilisera exactement comme avec UDP.

TCP : côté client : connect

La demande de connexion : **connect(2)**

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

DESCRIPTION

L'appel système `connect()` connecte la socket `sockfd` à l'adresse indiquée par `addr`. L'argument `addrlen` indique la taille de `addr`. Le format de l'adresse `addr` est déterminé par la famille de la socket `sockfd`...

VALEUR RENVOYÉE

`connect()` renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur.

Remarque : les arguments **addr** et **addrlen** sont ceux obtenus classiquement à l'aide de la fonction **getaddrinfo(3)**.

TCP : côté client : send

L'émission d'octets est réalisée via **send(2)** :

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

DESCRIPTION

L'appel `send()` ne peut être utilisé qu'avec les sockets connectées (ainsi, le destinataire visé est connu). La seule différence entre `send(2)` et `write(2)` est la présence de `flags`. Si `flags` est nul, `send(2)` est équivalent à `write(2)`.

VALEUR RENVOYÉE

En cas d'envoi réussi, ces fonctions renvoient le nombre de caractères envoyés. En cas d'erreur, -1 est renvoyé, et `errno` contient le code d'erreur.

Remarque : nous n'utiliserons pas de `flags` dans le cadre de ce cours. La valeur à préciser pour l'argument **flags** sera donc **0**.

TCP : côté client : recv

La réception d'octets est réalisée via **recv(2)** :

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

DESCRIPTION

L'appel `recv()` est normalement utilisé sur une socket connectée.

VALEUR RENVOYÉE

`recv()` renvoie le nombre d'octets reçus si elle réussit, ou -1 si elle échoue. La valeur de retour sera 0 si le pair a effectué un arrêt normal.

Remarque : si aucun **flag** ne doit être précisé (**flags** fixé à 0) il est possible de remplacer **recv(2)** par **read(2)**.

TCP : côté client : close

La terminaison de la connexion : **shutdown(2)/close(2)**

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

DESCRIPTION

La fonction `shutdown()` termine tout ou partie d'une connexion full-duplex sur la socket `sockfd`. Si `how` vaut `SHUT_RD`, la réception est désactivée. Si `how` vaut `SHUT_WR`, l'émission est désactivée. Si `how` vaut `SHUT_RDWR`, l'émission et la réception sont désactivées.

VALEUR RENVOYÉE

S'il réussit, cet appel système renvoie 0. S'il échoue, il renvoie -1 et remplit `errno` en conséquence.

Remarque : **close(s)** ; est équivalent à **shutdown(s, SHUT_RDWR)** ;

TCP : client complet

Nous avons toutes les briques pour construire notre client TCP !

Nous reprenons l'exemple d'une application client/serveur dont le serveur retourne ce qu'il lit en majuscules. Cependant pour mettre en évidence la notion de connexion notre client est capable d'émettre plusieurs lignes.

```
s=créer une socket TCP
adr=construire l'adresse du serveur
se connecter au serveur
tant que (ligne=lire une ligne) est vrai faire
    émettre la ligne au serveur
    attendre sa réponse
    afficher la réponse
fin tant que
fermer la connexion
```

TCP : client complet

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>

#include <unistd.h>      /* pour read(2)/write(2) */

#include <netdb.h>        /* pour getaddrinfo*/
#include <string.h>        /* pour memset */

#include <arpa/inet.h>    /* pour inet_ntop */

#define LINE_MAX 1024 /* taille MAX en réception */

void usage() {
    fprintf(stderr,"usage : client hostname port\n");
    exit(1);
}
```

TCP : client complet

```
int main(int argc, char **argv) {
    int s, ret;

    struct addrinfo hints, *result;

    char msg[LINE_MAX];
    char response[LINE_MAX];

    /* Vérification des arguments */
    if(argc!=3) {
        fprintf(stderr,"Erreur : Nb args !\n");
        usage();
    }
```

TCP : client complet

```
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_flags = 0;
hints.ai_family = AF_UNSPEC; /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_STREAM; /* Datagram socket */
hints.ai_protocol = 0; /* Any protocol */
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;

ret = getaddrinfo(argv[1], argv[2], &hints, &result);
if (ret != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
    exit(EXIT_FAILURE);
}
```

TCP : client complet

```
/* Cr ation de la socket IPv4/IPv6 */
if((s=socket(result->ai_family, result->ai_socktype,
              result->ai_protocol))==-1) {
    perror("socket"); exit(1);
}

/* Connexion au serveur */
if(connect(s, result->ai_addr, result->ai_addrlen)) {
    perror("connect"); exit(1);
}

freeaddrinfo(result); // si result ne sert plus
```

TCP : client complet

```
while(fgets(msg, LINE_MAX, stdin) != NULL) {  
  
    msg[strlen(msg)-1]=0; /* retire '\n' final */  
  
    /* Émission */  
    if(send(s, msg, strlen(msg)+1, 0) != strlen(msg)+1) {  
        perror("send"); exit(1);  
    }  
  
    /* Attente et lecture de la réponse */  
    ret=recv(s, response, LINE_MAX, 0);  
  
    if(ret==0) {  
        fprintf(stderr, "Déconnexion du serveur !\n");  
        exit(1);  
    }  
  
    else if(ret == -1) {  
        perror("recv"); exit(1);  
    }  
}
```

TCP : client complet

```
    /* Traitement de la réponse */
    puts(response);
}

if(close(s)==-1) {
    perror("close");
    exit(1);
}

return 0;
}
```

TCP : côté serveur

Définition de la taille de la file des connexions pendantes (**listen(2)**) :

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

DESCRIPTION

`listen()` marque la socket référencée par `sockfd` comme une socket passive, c'est-à-dire comme une socket qui sera utilisée pour accepter les demandes de connexions entrantes en utilisant `accept(2)`.

Le paramètre `backlog` définit une longueur maximale pour la file des connexions en attente pour `sockfd`. Si une nouvelle connexion arrive alors que la file est pleine, le client reçoit une erreur indiquant `ECONNREFUSED...`

VALEUR RENVOYÉE

S'il réussit, cet appel système renvoie 0. S'il échoue, il renvoie -1 et remplit `errno` en conséquence.

TCP : côté serveur

Accepter une connexion d'un client (**accept(2)**) :

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

DESCRIPTION

L'appel système `accept()` est employé avec les sockets utilisant un protocole en mode connecté (`SOCK_STREAM`, `SOCK_SEQPACKET`). Il extrait la première connexion de la file des connexions en attente de la socket `sockfd` à l'écoute, crée une nouvelle socket et alloue pour cette socket un nouveau descripteur de fichier qu'il renvoie. La nouvelle socket n'est pas en état d'écoute. La socket originale `sockfd` n'est pas modifiée ...

VALEUR RENVOYÉE

S'il réussit, cet appel système renvoie un entier positif ou nul, qui est un descripteur pour la socket acceptée. En cas d'erreur, il renvoie -1 et remplit `errno` avec le code d'erreur.

TCP : serveur complet

Voici l'algorithme du serveur passant en majuscules les messages en provenance du client (un seul à la fois):

```
construire l'adresse d'attachement  
créer la socket d'écoute  
attacher la socket d'écoute  
définir la taille de la file des connexions  
répéter à l'infini  
    socket_service = accepter une connexion  
    tant que (msg=lecture sur socket de service) est vrai faire  
        passage en majuscules  
        émission sur socket de service  
    fin tant que  
    fermer la socket de service  
fin répéter
```

TCP : serveur complet

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>
#include <unistd.h>      /* pour read(2)/write(2) */

#include <netdb.h>
#include <string.h>       /* pour memset */

#include <ctype.h>        /* pour toupper */

#include <arpa/inet.h>   /* pour inet_ntop */

#define REQUEST_MAX 1024    /* taille MAX en réception */

void usage() {
    fprintf(stderr, "usage : serveur port\n");
    exit(1);
}
```

TCP : serveur complet

```
int main(int argc, char **argv) {
    int s, sock, ret;

    struct addrinfo hints, *result;

    struct sockaddr_storage src_addr;
    socklen_t len_src_addr;

    char request[REQUEST_MAX];

    /* Vérification des arguments */
    if(argc!=2) {
        fprintf(stderr,"Erreur : Nb args !\n");
        usage();
    }
```

TCP : serveur complet

```
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_flags = AI_PASSIVE; /* Equiv INADDR_ANY */
hints.ai_family = AF_INET6; /* Allow IPv4 or IPv6 */
hints.ai_socktype = SOCK_STREAM; /* Flux => TCP */
hints.ai_protocol = 0; /* Any protocol */
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;

ret = getaddrinfo(NULL, argv[1], &hints, &result);
if (ret != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
    exit(EXIT_FAILURE);
}
```

TCP : serveur complet

```
/* Cr ation de la socket IPv4/IPv6 */
if((s=socket(result->ai_family, result->ai_socktype,
              result->ai_protocol))==-1) {
    perror("socket"); exit(1);
}

/* Attachement de la socket */

if (bind(s, result->ai_addr, result->ai_addrlen) == -1) {
    perror("bind"); exit(1);
}

freeaddrinfo(result);

/* d efinition de la taille de la file d'attente */

if(listen(s, 5)) {
    perror("listen"); exit(1);
}
```

TCP : serveur complet

```
while(1) { /* boucle du serveur */

    /* Attente d'une connexion */

    puts("En attente de connexion...");

    len_src_addr=sizeof src_addr;
    if((sock=accept(s, (struct sockaddr *)&src_addr,
        &len_src_addr))==-1) {
        perror("accept"); exit(1);
    }

    puts("Connexion acceptée !");
```

TCP : serveur complet

```
/* boucle de traitement du client */

while((ret=recv(sock, request, REQUEST_MAX, 0))>0) {

    request[ret]=0;

    /* traitement de la requête(passage en majuscule) */
    {
        int i=0;

        while(request[i]) {
            request[i]=toupper(request[i]);
            ++i;
        }
    }
}
```

TCP : serveur complet

```
/* Émission de la réponse */

if(send(sock, request, strlen(request)+1, 0) !=  
    strlen(request)+1) {  
    perror("send"); exit(1);  
}  
} /* fin de la boucle de traitement du client */

if(close(sock)==-1) {  
    perror("close"); exit(1);  
}

fprintf(stderr, "Fin de connexion !\n");
if(ret==-1) {
    perror("recv");
}

} /* fin boucle principale du serveur */

return 0;
}
```

TCP : remarques

- Lorsqu'une connexion est "brutalement fermée" du côté du serveur, le port associé à la socket d'écoute est inutilisable pendant un délai de plusieurs minutes pour assurer que le client soit averti de la terminaison de la connexion.
 - Le message **bind: Address already in use** est affiché si le serveur essaye de s'attacher trop rapidement au même port. Patientez ou changez de port !
- Pour observer l'état des sockets TCP de votre système vous pouvez exécuter :
 - **netstat -atp**
- Dans l'algorithme (et le programme) vous avez pu constater que le serveur ne pouvait gérer qu'un client à la fois. Proposez une piste (ou plusieurs) pour pouvoir gérer plusieurs clients simultanément.