IV.1. Généralités, Sockets en C

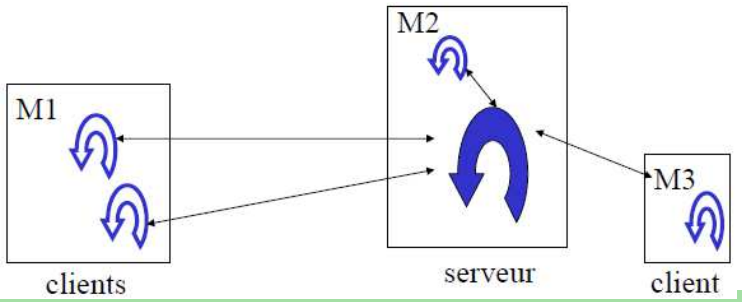
L'objectif des sockets est de permettre à des processus distants de communiquer sur le réseau

Il est aussi possible d'utiliser les sockets pour faire communiquer des processus locaux mais d'autres outils existent (Pipes, IPC)

Chaque socket, cliente ou serveur, est adressé sur le réseau par son adresse IP et son numéro de port

Le canal de communication est bidirectionnel, chaque socket pouvant émettre et recevoir des messages. L'échange est bloquant (envoi d'un message et attente de la réponse)

C'est la socket cliente qui initie le dialogue par une demande (elle est active).
La socket serveur se met en attente (elle est passive).



On peut utiliser les sockets en C, en Java, en Perl …

Les données pouvant être codées différemment de part et d'autre sur le réseau (cela peut notamment dépendre du processeur de chaque machine impliquée dans la communication), une conversion de format (format local <-> format commun) peut donc être nécessaire.

Dans le modèle OSI, les sockets se situent juste au dessus de la couche transport. Cette conversion de format ne peut donc être effectuée par un protocole de la couche présentation (tel que ASN …). Elle doit donc être faite dans le programme lui même.

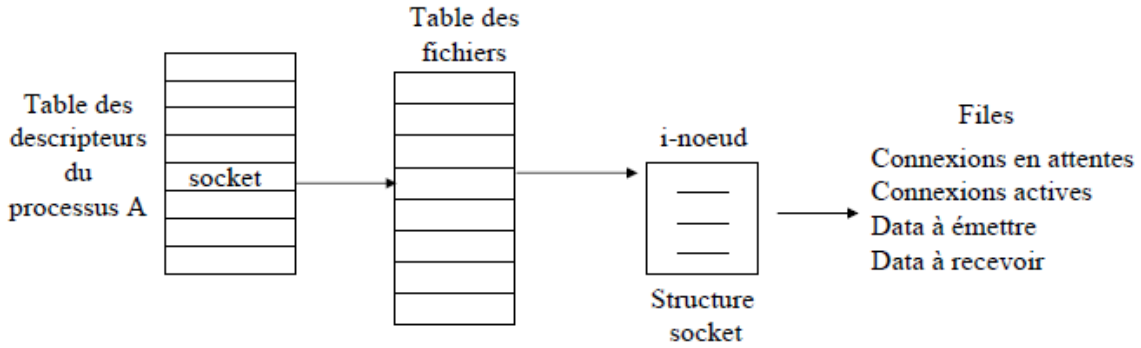
La communication peut être établie en mode connecté (sockets basés sur TCP) ou en mode non connecté (sockets basés sur UDP).

En mode non connecté, il est possible de faire de la multidiffusion, en broadcast ou en multicast.

Le serveur peut travailler en mode itératif (les clients sont traités l'un après l'autre) ou en mode concurrent (plusieurs clients sont traités en même temps).

En mode concurrent, un processus d'écoute (ou veilleur) créé en général un nouveau processus (exécutant) pour gérer chaque nouveau client. Les exécutants peuvent être des processus lourds (fork) ou légers (threads)

Descripteur : C'est la référence de chaque socket stockée dans la table des descripteurs.



Côté client et côté serveur : Le descripteur de la socket locale peut être laissée à la charge de son processus créateur (système d'exploitation).

Côté client : La socket cliente doit connaître les informations d'adressage (nom/@IP, port) de la socket serveur pour pouvoir l'appeler.

Côté serveur : Les informations d'adressage de la socket cliente sont transmises automatiquement lors de l'appel. La socket serveur peut alors lui répondre.

Côté serveur en mode concurrent avec fork() : le processus fils hérite automatiquement des descripteurs (donc des sockets) du père. Il ne faut pas oublier de fermer la socket fille dans le processus de la socket mère (inutile) et la socket mère dans le processus de la socket fille (inutile également).

Propriétés d'une socket

A chaque socket sont associés en plus du descripteur :

- Un domaine : Local si les processus communicants sont situés sur la même machine (AF_UNIX), ou Internet si les processus communiquent à travers le réseau Internet (AF_INET) ; Autres (AF_NS, AF_SNA, AF_APPLETALK)

- Un type : Mode déconnecté (SOCK_DGRAM), Mode connecté (SOCK_STREAM) Autres (SOCK_RAM, SOCK_SEQPACKET …)

Tampons de lecture (entrée) et d'écriture (sortie)

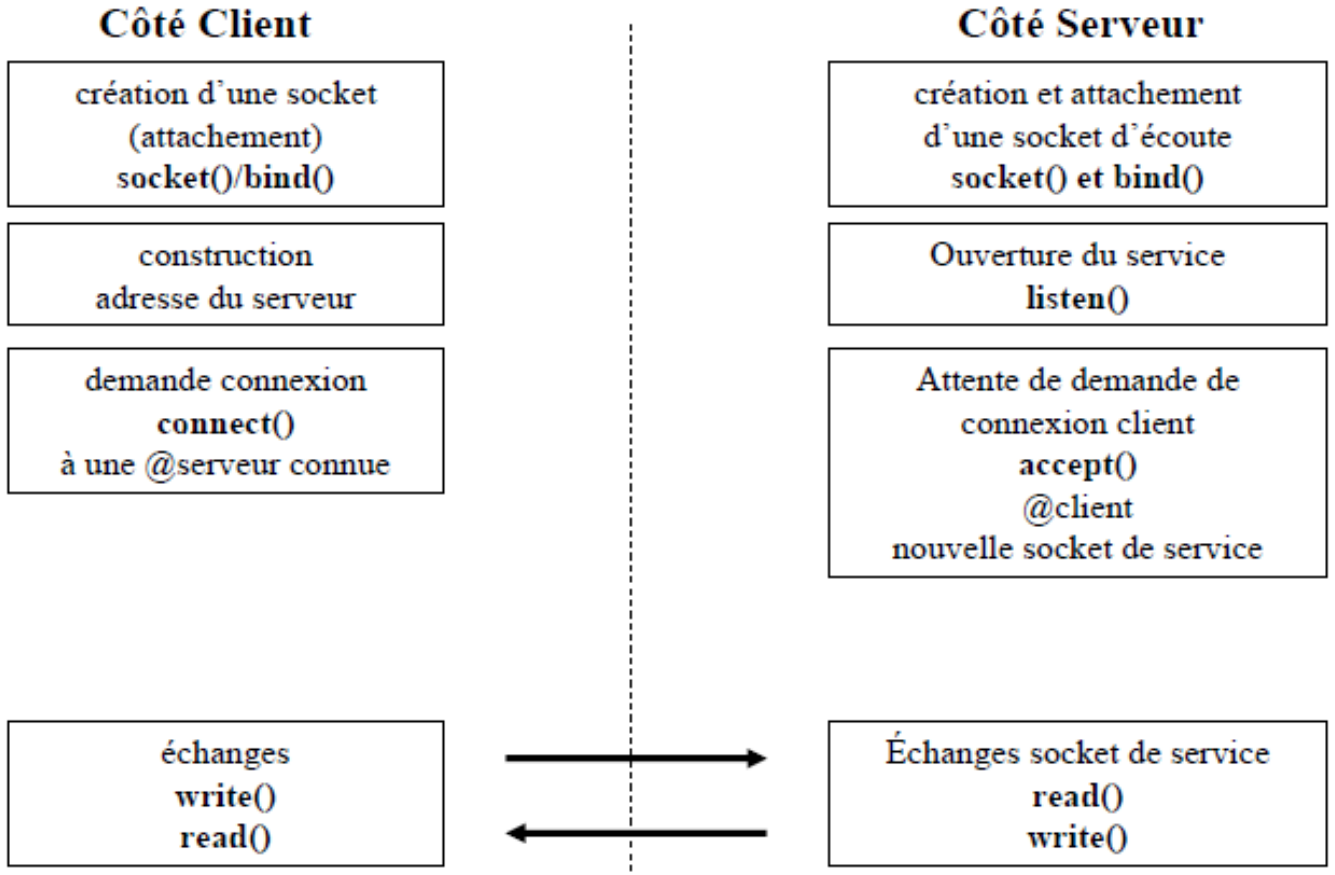
Une socket est associée à deux mémoires tampons gérées par le système : un tampon pour l'émission et un tampon pour la réception.

Ecriture : Le système copie les données d'une variable (espace mémoire du processus) vers le tampon d'émission. L'écriture est bloquante sur tampon plein (le processus est endormi en attendant que le tampon d'émission se vide). L'écriture s'achève à la fin de la copie des données dans le tampon d'émission.

Lecture : Le système recopie les données du tampon de réception dans l'espace alloué pour la variable du processus. Par défaut, la lecture est bloquante sur tampon vide (il faut attendre qu'il se remplisse) mais il est possible de rendre la lecture non bloquante.

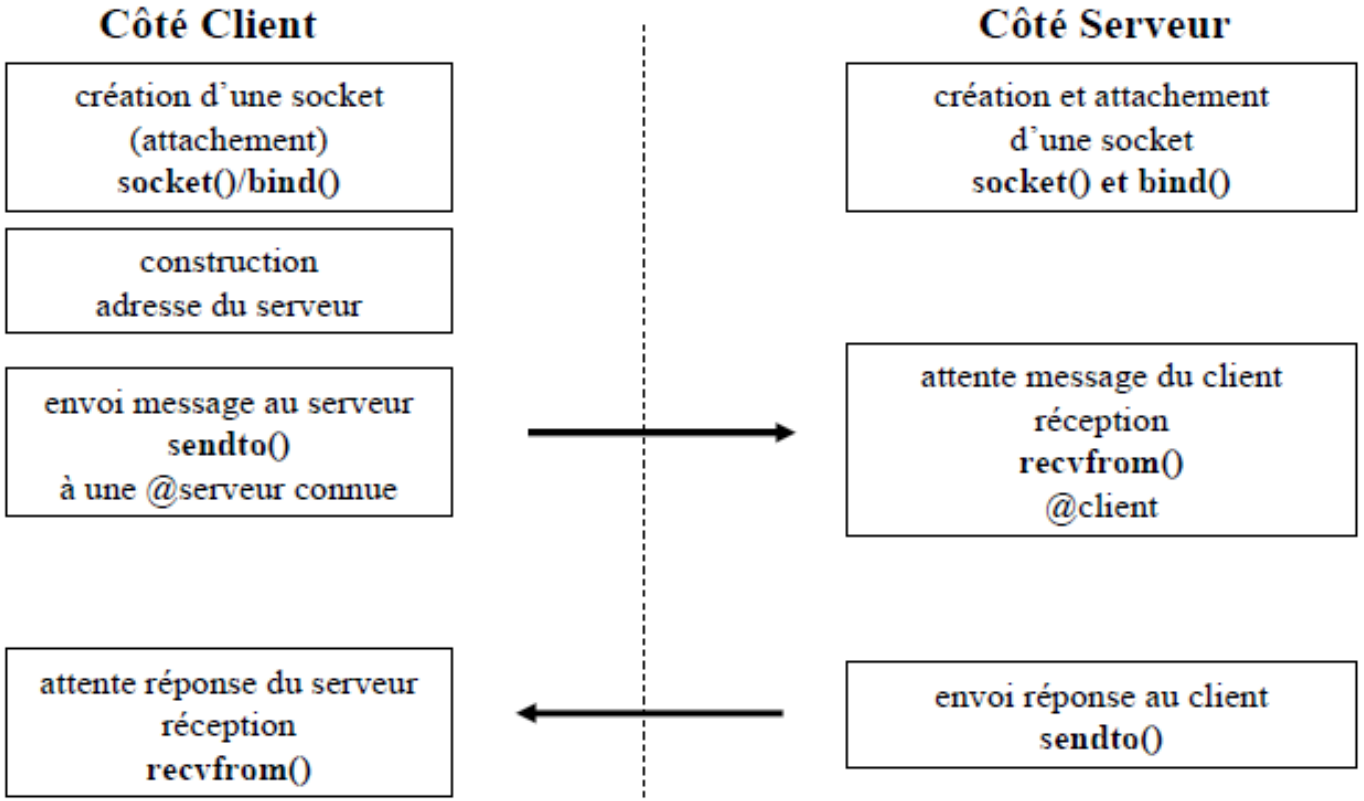
Sockets en mode connecté

Communication connectée, fiable, ordonnée, gestion de priorités, pas de multi-diffusion.



Sockets en mode non connecté : UDP

Communication non connectée, non fiable, non ordonnée, plus légère, multidiffusion possible.



Les bibliothèques

```
#include <sys/types.h> /* type  u_short pour unsigned short, etc. */
#include <sys/socket.h> /* fonctions socket(...), bind(...), accept(...), etc. */
#include <netinet/in.h> /* types pour l'adressage : in_port_t pour u_short, etc. */
#include <arpa/inet.h> /* fonctions de conversion : inet_pton(...), inet_ntoa(...), etc. */
#include <netdb.h> /* fonctions gesthostbyaddr(...), gesthostbyname(...), etc.
structure hostent etc. */
#include <unistd.h> /* fork(...), getpid(...), sleep(...), pause(...), etc. */
```

Les types de données

```
struct sockaddr { /* adressage */
    unsigned short sa_family;  /* famille (de protocole pour cette adresse) */
    char sa_data[14];  /* valeur de l'adresse sur 14 octets */
};

struct sockaddr_in { /* adressage sur internet */
   short sin_family; /* famille (de protocole pour cette adresse) : AF_INET en gal */
   unsigned short sin_port; /* numéro de port (0 si port non utilisé) */
   struct in_addr sin_addr; /* adresse IP */
   char sin_zero[8]; /* un champ de 8 zéros */
};

struct in_addr { /*adresse IP */
   unsigned long s_addr; /* adresse IP, si INADDR_ANY le système choisit */
};
```

```
struct sockaddr_in6 { /* adressage avec des adresses IP version 6 */
    u_int16_t sin6_family;
    u_int16_t sin6_port; /* numéro de port */
    u_int32_t sin6_flowinfo;
    struct in6_addr sin6_addr; /* adresse IPv6 */
    u_int32_t sin6_scope_id;
};


struct in6_addr {
    unsigned char s6_addr[16]; /* adresse Ipv6 (sur 16 octets) */
};

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 */
};

#define h_addr  h_addr_list[0] /* pour compatibilité */
```

Les primitives de la couche Socket

Création d'une socket

int socket(int domain, int type, int protocol); /* Création d'une socket (allocation d'un descripteur de socket local). domain : AF_INET ou AF_INET6 pour une communication sur internet. type : SOCK_DGRAM (sockets déconnectées) ou SOCK_STREAM (sockets connectées). protocol : laisser à 0 pour laisser le système choisir le bon (SOCK_STREAM => TCP, SOCK_DGRAM => UDP). Retourne un descripteur de socket ou -1 en cas d'erreur (avec errno mis à jour). */

Liaison d'une socket à une adresse

int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen); /* Liaison d'une socket à une adresse IP et un port. sockfd : descripteur de socket local renvoyé par la fonction socket(...). addr : pointeur sur la structure de type sockaddr contenant les informations d'adressage local. addrlen : taille de la structure précédente. La fonction retourne 0 si succès ou -1 si erreur (avec errno mis à jour). */

Etablir une connexion avec un serveur

int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen); /*Connecter la socket locale à une socket serveur. sockfd est le descripteur de socket local. serv_addr est le pointeur sur la structure contenant les informations d'adressage du serveur. addrlen donne la taille de la structure précédente. Retourne 0 si connexion réussie ou -1 si erreur (avec errno mis à jour). */

Mettre une socket serveur en attente de connexion

int listen (int sockfd, int nbc); /* mettre une socket serveur en attente de demande de connexion (socket d'écoute). sockfd est le descripteur de socket local. nbc indique le nombre de connexions en attente max avant que le nouvelles soient rejetées. Retourne 0 si succès ou -1 si erreur (avec errno mis à jour). */

Accepter une demande de connexion de la part d'un client

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen); /* Accepte une demande de connexion entrante sur la socket d'écoute. sockfd est le descripteur de socket local, addr pointe sur la structure qui sera mise à jour avec les informations d'adressage de la socket cliente. addrlen : pointeur sur l'entier qui va indiquer la taille de la structure précédente. Retourne le descripteur de la socket de service (exécutant) ou -1 si erreur (avec errno mis à jour). */

Envoi d'un message en mode déconnecté

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen); /* Envoi d'un message en mode déconnecté. sockfd est le descripteur de socket local. buf est le pointeur sur les données à envoyer (buffer d'envoi). len donne le nombre d'octets (max) à envoyer. flags : voir send(...). to pointe sur la structure contenant les informations d'adressage de la socket distante (destinataire), tolen indique la taille de la structure précédente. Retourne le nombre d'octets effectivement envoyés ou -1 si erreur (avec errno mis à jour). */

Réception en mode déconnecté

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *, from, socklen_t *fromlen); /* Réception en mode déconnecté. sockfd est le descripteur de socket local. buf est le pointeur sur le buffer de réception et len le nombre d'octets (max) à lire. flags : voir recv(...). from pointe sur la structure qui sera mise à jour avec les informations de la socket distante (émettrice). fromlen pointe sur la taille de la structure précédente. Retourne le nombre d'octets effectivement reçus ou -1 si erreur (avec errno mis à jour). */

Emission en mode connecté

ssize_t send(int sockfd, const void *buf, size_t len, int flags); /* Emission d'un message en mode connecté. sockfd est le descripteur de socket local (celui de la socket de service si côté serveur). buf est un pointeur sur les données à envoyer. len indique le nombre d'octets à envoyer. flags permet de spécifier des options d'envoi. Mettre le flag à 0 pour un envoi "normal" ; sinon MSG_OOB (envoi avec une priorité supérieure : le récepteur recevra un signal SIGURG), MSG_DONTWAIT (permettre un envoi non bloquant), MSG_NOSIGNAL (habituellement, si le récepteur n'est pas prêt à recevoir, un signal SIGPIPE est typiquement reçu : ce n'est pas le cas avec cette option), etc. Retourne le nombre d'octets effectivement envoyés ou -1 si erreur (avec errno mis à jour). Si la socket distante a été fermée, la fonction renverra la valeur -1 + réception d'un signal SIGPIPE (sauf si le flag MSG_NOSIGNAL est positionné). */

Réception en mode connecté

ssize_t recv(int sockfd, void *buf, size_t len, int flags); /* Réception d'un message en mode connecté. sockfd est le descripteur de socket local (socket de service côté serveur). buf est un pointeur sur le buffer de réception. len indique le nombre d'octets (max) à lire. flags peut être mis à 0 pour une réception classique ; sinon MSG_OOB (réception prioritaire, qui correspond à un envoi prioritaire, c'est à dire un send(....) avec le flag MSG_OOB ; remarque : on peut aussi définir un handler pour le signal SIGURG), MSG_PEEK (lire les données en attente sans les prélever : une lecture ultérieure donnera le même résultat), MSG_WAITALL (bloque l'opération de lecture jusqu'à ce que la taille des données à lire correspondent au paramètre len ; le blocage peut toutefois être interrompu si signal d'interruption ou si une erreur ou une déconnexion se produisent), etc. Retourne le nombre d'octets effectivement reçus ou -1 si erreur (avec errno mis à jour). */

Fermeture d'une socket

int close(int sockfd); /* Ferme la socket (à faire une fois tous les envois et/ou réceptions terminé(e)s. sockfd est le descripteur de socket local. Après le close(...) local, un appel à recv(...) de la socket distante retournera 0, un appel à send(...) retournera -1 (plus un signal SIGPIPE sera reçu et errno sera mis à jour avec la valeur EPIPE). Retourne 0 si fermeture OK ou -1 si erreur (avec errno mis à jour). */

int shutdown(int sockfd, int how); /* Fermeture d'une socket dans un sens. En entrée si how vaut SHUT_RD, en sortie (SHUT_WR) ou en entrée/sortie (SHUT_RDWR). Retourne 0 si succès ou -1 si erreur (avec errno mis à jour) */

int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen); /* Modifier une option (caractéristiques) d'une socket. sockfd est le descripteur de socket. level désigne un niveau : SOL_SOCKET (modifications au niveau socket), IPPROTO_IP (niveau IP), IPPROTO_TCP (niveau TCP). optname désigne l'option à mettre à jour : par exemple SO_BINDTODEVICE (associe la socket à une interface physique comme eth0 à la place d'utiliser bind(...) qui la lie à une adresse IP), SO_REUSADDR (permet de lier d'autres sockets au même port même si une socket est en cours d'écoute sur ce port, SO_TYPE (changer le type de la socket), SO_BROADCAST (permet d'envoyer et de recevoir des datagrammes IP depuis/vers une adresse broadcast), SO_RCVBUF (mettre à jour le tampon de réception de la socket), SO_RCVSEND (mettre à jour le tampon d'envoi). optval pointe typiquement sur un int qui indique la valeur de l'option en question (pour les booléens, 0 vaut false et une valeur différente de 0 vaut true) ; optval peut pointer sur null si aucun paramètre n'est passé. optlen vaut typiquement sizeof(int). La fonction retourne 0 si succès ou -1 si erreur (avec errno mis à jour). */

int getsockopt(int sockfd, int level, int optname, const void *optval, socklen_t *optlen); /* Même chose mais pour récupérer la valeur d'une option (optval pointe sur la valeur récupérée et optlen sur sa taille). */

int getsockname (int sockfd, struct sock_addr *addr, socklen_t *addrlen); /* Après un bind (…), cette fonction permet de récupérer l'adresse effective de la socket locale dans une structure de type sock_addr (pointée par addr). La taille de cette adresse est stockée à l'adresse pointée par addrlen. Retourne 0 si succès ou -1 si erreur (avec errno mis à jour). */

Récupération des informations d'adressage

int gethostname(char *name, size_t len); /* Récupère le nom d'hôte local. Le paramètre name pointe sur le buffer qui contiendra le nom d'hôte, et len indique la taille du buffer. Retourne 0 si succès, -1 si erreur (avec errno mis à jour). */

struct hostent *gethostbyname(const char *name); /* A partir d'une chaîne contenant un nom d'hôte (ou une adresse IP, IPv4 en notation décimale pointée ou IPv6 en notation RFC 1884), retourne un pointeur sur une structure de type hostent. Retourne NULL si erreur (avec h_errno mis à jour). */

struct hostent *gethostbyaddr(const void *addr, int len, int type); /*. A partir d'un pointeur sur une valeur binaire de taille len et de type AF_INET ou AF_INET6, retourne une structure de type hostent. La valeur binaire pointée par le paramètre addr peut être contenue dans une structure de type in_addr (IPv4 : taille 4 octets) ou in6_addr (IPv6 : taille 16 octets). Retourne NULL si erreur (avec h_errno mis à jour). */

Remarque : ces deux dernières fonctions sont deprecated : voir aussi les fonctions **getaddrinfo(...)** et **getnameinfo(...)**

Manipulation d'adresses IP

char *inet_ntoa (struct in_addr in); /* Convertit une adresse IP contenue dans une structure de type in_addr (typiquement un entier long) en chaîne */

int inet_aton(const char *cp, struct in_addr *inp); /* Convertit une adresse IP sous forme de chaîne en structure de type in_addr. Retourne une valeur non nulle si l'adresse est valide et 0 sinon */

const char *inet_ntop(int type, const void *src, char *dst, socklen_t size); /* convertit une adresse IP contenue dans une structure de type in_addr ou in6_addr en chaîne. Le paramètre type vaut AF_INET ou AF_INET6. Le paramètre src pointe sur une structure de type in_addr ou in6_addr. Le paramètre dst est le pointeur sur la chaine destination et size la taille maximale de cette chaîne. Retourne un pointeur sur la chaîne destination si succès et NULL si erreur (avec errno mis à jour). */

int inet_pton (int type, const char *src, void *dest); /* convertit une adresse IP sous forme de chaîne en une valeur contenue dans une structure de type in_addr ou in6_addr. Le paramètre type doit-être AF_INET ou AF_INET6. Le paramètre src pointe sur la chaîne. Le paramètre dest pointe sur la structure de type in_addr ou in6_addr où le résultat sera placé. La fonction retourne 1 si tout s'est bien passé, -1 si il y a une erreur (avec errno mis à jour) et 0 si l'adresse IP n'est pas valide. */

Conversion de format

La représentation des données peut être différente selon les machines communicantes sur le réseau. Ainsi, certaines informations fournies par une socket locale (adresse IP, numéro de port) peuvent être mal interprétées par la socket distante.

Il est alors nécessaire de définir un format réseau (ou format commun) afin d'éviter toute erreur d'interprétation de part et d'autre. Les données sont alors converties du format local vers le format réseau avant envoi, puis sont reconvertis du format réseau vers leur format local à la réception.

Fonctions préfixées par "hton" ("host to network") ou "ntoh" ("network to host") .

unsigned long ntohl (unsigned long netlong); /* conversion d'un entier long du format réseau vers le format local */
unsigned short ntohs (unsigned short netshort); /* entier short du format réseau vers le format local */
unsigned long htonl (unsigned long hostlong); /*entier long du format local vers le format réseau */
unsigned short htons (unsigned short hostshort); /* entier short du format local vers le format réseau */

Exemple : Motorola, IBM : codage big endian, Intel : codage little indian. La conversion en format réseau consiste en une conversion en big endian.

#IP	Valeur Binaire	Format Endian
132.227.70.77	84 C3 46 4D	Big (standard)
132.227.70.77	4D 46 C3 84	Little

Gestion des erreurs

int errno; /* contient le code d'erreur du dernier appel système qui a planté. Sa valeur n'est significative que lorsque l'appel système a échoué (généralement en renvoyant -1), car même en cas de réussite, une fonction de bibliothèque peut modifier errno. Le fichier errno.h contient une liste de constantes symboliques pour les erreurs : EACCES 13 : Permission refusée, ECONNREFUSED 111 : Connexion refusée, EADDRINUSE 112 : Adresse en cours d'utilisation etc. */

void perror(const char *s); /* De nombreuses fonctions retournent - 1 et mettent à jour errno avec un code d'erreur (nombre). perror(...) permet de traduire les codes d'erreur en une forme intelligible. Le paramètre s permet de rajouter une description avant l'affichage du message d'erreur (il peut aussi être mis à NULL). L'affichage sur la sortie erreur standard concerne d'abord la chaine pointée par s, suivie de ":" suivie du message d'erreur. */

char *strerror(int errnum); /* Retourne le message d'erreur associé au code d'erreur passé en paramètre (errnum) */

Quelques fonctions C diverses (pouvant être utiles pour le TD)

void *memcpy(void *dest, const void *src, size_t n);
/* Copie n octets de l'adresse source src vers l'adresse destination dest. Retourne cette adresse destination. */

char *strcpy (char *dest, const char *src);
/* Copie la chaîne pointée par src à l'adresse destination dest. Retourne cette adresse destination. */

void *memset(void *str, int c, size_t n);
/* Copie la valeur c sur n octets à l'adresse str. Retourne cette adresse. Remarque : c est passé en tant que int mais la fonction le traite comme un unsigned char. */

int strcmp(const char *ch1, const char *ch2);
/* Compare la chaîne pointée par ch1 à la chaîne pointée par ch2. Retourne une valeur inférieure à 0 si ch1 est inférieure à/avant ch2, supérieure à 0 si ch1 est supérieure à/après ch2 et égale à 0 si les deux chaînes sont égales. */

int toupper(int c);
/* Convertit une lettre de minuscule à majuscule. Si l'argument n'est pas un caractère de l'alphabet convertible en majuscule, il est laissé tel quel. La valeur est passée et retournée en tant que int mais traitée comme un unsigned char. */

int atoi(const char *str);
/* Convertit une chaîne en int (la chaîne doit bien sûr être convertible : "12" par exemple). Si la chaîne n'est pas convertible, la fonction retourne 0. */

IV.2. Sockets en Java

Les packages

java.net : contient l'ensemble des classes pour programmer des sockets (ServerSocket, Socket …)

java.io : contient l'ensemble des classes pour gérer les flots (ou flux) d'entrée/sortie (InputStream, InputStreamReader …)

Les classes et leurs méthodes

Classe **ServerSocket** : serveur d'écoute (pour les sockets connectées donc)

```
public ServerSocket ();
public ServerSocket (int port);
public ServerSocket (int portLoc, int backlog);
public ServerSocket (int portLoc, int backlog, InetAdress addrLoc);
/* Constructeurs de la classe. Le paramètre portLoc indique le numéro de port local
(laisser à 0 pour allocation automatique), backlog la taille de la file d'attente (nombre
de clients maxi en attente : 50 par défaut) et addrLoc permet de désigner une
adresse IP spécifique si la machine en possède plusieurs (sinon choix automatique).

void bind (socketAdress addrLoc);
void bind (socketAdress addrLoc, int backlog);
/* Permet de lier le serveur d'écoute à une socket (couple @IP/port) spécifique. */
```

public Socket accept()
/* Mise en attente des demandes de connexion entrantes. A chaque nouvelle demande, création d'une socket (de service) pour la traiter. Cette méthode est bloquante mais l'attente peut-être limitée par l'appel de la méthode setSoTimeOut(...) */

public void setSoTimeout(int timeout);
/* Met à jour la valeur de SO_TIMEOUT avec un délai indiqué en millisecondes. Déblocage de la méthode accept() une fois ce délai atteint (cette méthode doit être appelée avant le accept()). Quand le délai est atteint, une java.net.SocketTimeoutException est levée. Le délai doit être supérieur à 0. Un délai mis à 0 est interprété comme un délai infini (cela peut servir à désactiver un délai précédemment établi). */

public void setReceiveBufferSize(int size);
/* Modification de la taille du buffer de réception (mise à jour de SO_RCVBUF + de la taille de la fenêtre TCP qui prévient l'émetteur de ce qu'il peut émettre) */

public void close();
/* Fermer la socket. Chaque demande de connexion en attente sur accept() lèvera une SocketException */

Classe **Socket** : Pour les sockets connectées. Côté client, un objet de type Socket sera créé par appel au constructeur ; côté serveur, il sera obtenu par retour de la méthode accept()).

public Socket ();
public Socket (InetAddress addrDist, int portDist);
public Socket (InetAddress addrDist, int portDist, InetAddress addrLoc, int portLoc);
/* Constructeurs de la classe. Créé une socket. addrDist et portDist sont les informations d'adressage de la socket serveur à laquelle cette socket se connecte. addrLoc et portLoc sont les informations d'adressage local */

void bind (socketAdress addrLoc);
/* Permet de lier la socket à ses informations d'adressage (@IP/port). */

public void connect(SocketAddress addrDist);
public void connect(SocketAddress addrDist, int timeout) ;
/* Connecte la socket au serveur. Un délai d'attente maximum permet de ne pas rester bloqué indéfiniment si le serveur est occupé */

public void setSendBufferSize(int size);
/* Modification de la taille du buffer d'envoi (mise à jour de SO_SNDBUF avec la valeur en octets fournie (>0) */

public void setReceiveBufferSize(int size);
/* Modification de la taille du buffer de réception (mise à jour de SO_RCVBUF avec la valeur en octets fournie (>0) */

InputStream public getInputStream();
/* Ouvre un flux d'entrée pour cette socket */

OutputStream public getOutputStream();
/* Ouvre un flux de sortie pour cette socket */

public void setSoTimeout(int timeout);
/* Met à jour SO_TIMEOUT avec un délai indiqué en millisecondes. L'attente du flux d'entrée va alors se débloquer une fois ce délai atteint (cette méthode doit être appelée avant l'appel de la méthode bloquante). Quand le délai est atteint, une java.net.SocketTimeoutException est levée. Le délai doit être supérieur à 0. Un délai mis à 0 est interprété comme un délai infini (cela peut servir à désactiver un délai précédemment établi). */

public InetAddress getInetAddress();
/* Retourne l'@IP de la socket distante (à laquelle la socket locale est connectée). */

public void shutdownInput();
/* Ferme le flux d'entrée. Une lecture ultérieure retournera EOF. */

public void shutdownOutput();
/* Ferme le flux de sortie. Une écriture ultérieure lèvera une IOException */

public int getPort();
/* Retourne le numéro de port de la socket distante*/

public void close();
/* Ferme la socket */

Classe abstraite **InputStream** : Gère les flots d'octets entrants. Un objet de type InputStream peut être obtenu au départ par l'appel de la méthode getInputStream() de la classe Socket.

public InputStream();
/* Constructeur. Pas besoin à priori */

public abstract int read();
/* Lecture d'un octet. La valeur retournée est comprise entre 0 et 255. La méthode est bloquante. Si l'octet ne peut être lu (fin de flot), la méthode retourne -1. */

public int read(byte[] b);
/* Lecture d'un flot d'octets entrant et enregistrement dans b. La méthode retourne le nombre d'octets effectivement lus. La méthode est bloquante. Si aucun octet ne peut être lu, la méthode retourne -1. */

public int read(byte[] b, int offset, int len);
/* Lecture de len octets sur le flot entrant et enregistrement dans le tableau b à la position offset. */

public long skip(long n);
/* Saute n octets dans la lecture du flot entrant. Retourne le nombre d'octets effectivement sauté (qui peut être inférieur à n, voire nul, si la fin du flot est atteinte par exemple). */

public void close();
/* Ferme le flot entrant et relâche les ressources associées. En pratique, la méthode close() de InputStream ne fait rien. */

Classe **InputStreamReader** : Hérite de la classe **Reader** (classe qui gère les flots de caractères entrants). Elle permet de transformer un flot d'octets entrant en flot de caractères entrant en utilisant le codage désigné ou celui par défaut du système.

public InputStreamReader(InputStream in);
public InputStreamReader(InputStream in, String charsetName);
...
/* Constructeur. Créé un flot de caractères entrant en interprétant le flot d'octets entrant avec un codage de caractères (celui désigné en paramètre ou celui par défaut). */

public int read();
/* Lecture d'un caractère. Méthode bloquante. Retourne -1 si le caractère n'a pas pu être lu (fin de flot). */

public int read(char[] cbuf, int offset, int len);
/* Lit len caractères depuis le flot entrant et les place dans cbuf à la position offset. Retourne le nombre de caractères effectivement lus ou -1 */

public void close();
/* Ferme le flot entrant et relâche les ressources associées. */

Classe **BufferedReader** : Hérite également de **Reader**. Elle permet de lire dans un flot de caractères entrant tamponné.

public BufferedReader(Reader in);
public BufferedReader(Reader in, int sz);
/* Constructeurs. Créé un buffer d'entrée avec une taille par défaut (premier constructeur) ou bien avec la taille désignée (deuxième constructeur). */

public int read();
/* Lecture d'un caractère. Méthode bloquante. Retourne -1 si aucun caractère ne peut être lu (fin de flot). */

public int read(char[] cbuf, int offset, int len);
/* Lit len caractères sur le flot entrant et les place dans cbuf à la position offset. Retourne le nombre de caractères effectivement lus ou -1. */

public String readLine();
/* Lit une ligne (une ligne se termine par un caractère de fin de ligne ('\n', '\r' ou '\r\n') sur le flot entrant et la retourne dans une chaîne qui ne contient pas ce(s) caractère(s) de fin de ligne. La méthode retourne null si la fin du flot est atteinte. ***/**

public long skip(long n);
/* saute n caractères sur le flot entrant. Retourne le nombre de caractères effectivement sauté */

public void close();
/* Ferme le flot entrant et relâche les ressources associées. */

Classe abstraite **OutputStream** : Classe abstraite qui gère les flots d'octets sortants. Un objet de type OutputStream peut être au départ obtenu par l'appel de la méthode getOutputStream() de la classe Socket.

public OutputStream();
/* Constructeur. Pas besoin à priori. */

public abstract void write(int b);
/* Ecrit un octet b dans le flot sortant (l'octet en question est fourni comme un int : les 3 octets de poids fort de l'entier sont ignorés). */

public void write(byte[] b);
/* Ecrit b.length octets du tableau b vers le flot sortant. */

public void write(byte[] b, int offset, int len);
/* Lit len octets à la position offset dans le tableau b et les écrit dans le flot sortant. */

public void flush();
/* Vide le flot sortant en forçant l'envoi des données. */

public void close();
/* Ferme le flot sortant et relâche les ressources associées. En pratique, la méthode close() de OutputStream ne fait rien. */

Classe **OuputStreamWriter** : Hérite de **Writer** (classe qui gère les flots de caractères sortants). Elle permet de transformer un flot de caractères sortant en flot d'octets sortant en utilisant le codage désigné ou celui par défaut du système.

public OutputStreamWriter(OutputStream out);
public OutputStreamWriter(OutputStream out, String charsetName);
...
/* Constructeur. Créé un flot d'octets sortant en interprétant le flot de caractères sortant avec un codage de caractères (celui désigné en paramètre ou celui par défaut). */

public void write(int c);
/* Ecrit un caractère dans le flot sortant. */

public void write(char[] cbuf, int offset, int len);
/* Lit len caractères à partir de la position offset du tableau cbuf et les écrit dans le flot sortant. */

public void flush();
/* Vide le flot sortant. */

public void close();
/* Ferme le flot sortant, après l'avoir vidé. */

Classe **BufferedWriter** : Hérite également de **Writer**. Elle permet d'écrire dans un flot de caractères sortant tamponné.

public BufferedWriter(Writer out);
public BufferedWriter(Writer out, int sz);
/* Constructeurs. Créé un buffer de sortie avec une taille par défaut (premier constructeur) ou bien avec la taille spécifiée (deuxième constructeur) . */

public void write(int c);
/* Ecrit un caractère dans le flot sortant. */

public void write(char[] cbuf, int offset, int len);
/* Lit len caractères à partir de la position offset dans le tableau cbuf et les écrit dans le flot sortant. */

public void write(String s, int offset, int len);
/* Même chose mais la lecture se fait à partir d'une chaîne (s). */

public void newLine();
/* Ecrit un séparateur de ligne sur le flot sortant (plus sûr que '\n' qui dépend des systèmes). */

public void flush();
/* Vide le flot sortant en forçant l'envoi des données. */

public void close();
/* Ferme le flot sortant, après l'avoir vidé. */

Classe **PrintStream** : Hérite (indirectement) de **OutputStream**. Elle permet une interprétation des différents formats de données (int, float ...) pour écriture sur le flot d'octet sortant. Elle permet aussi optionnellement de spécifier une option de vidage automatique du tampon de sortie.

public PrintStream(OutputStream out);
public PrintStream(OutputStream out, boolean autoflush);
public PrintStream(File file);
...
/* Constructeurs. On peut définir un vidage automatique du tampon après chaque écriture (si autoflush vaut true). */

public void print(*type* x);
/* Avec *type* : char, boolean, int, long, float, double, String, char[] ou Object. Conversion puis écriture dans le flot sortant. */

public void println(*type* x);
/* Avec *type* : char, boolean, int, long, float, double, String, char[] ou Object. Même chose en ajoutant un séparateur de ligne. */

etc.

Classe **DatagramSocket** : Pour les sockets déconnectées.

public DatagramSocket ();
public DatagramSocket (int portLoc);
public DatagramSocket (int portLoc, InetAddress addrLoc);
public DatagramSocket (SocketAddress addrLoc);
/* Constructeur de la classe. On peut laisser le système récupérer l'adresse IP (ou en choisir une parmi plusieurs possibles) ou bien définir l'adresse à utiliser. On peut le laisser choisir un numéro de port (parmi ceux disponibles) ou bien définir ce numéro de port. */

public void bind(SocketAddress addrLoc);
/* Lie la socket à une adresse IP et un numéro de port */

public void send (DatagramPacket p);
/* Envoi d'un datagramme. Ce datagramme comprend, en plus des données (et des informations sur ces données : taille, offset), des informations concernant l'hôte distant (adresse IP, numéro de port). */

public void receive (DatagramPacket p);
/* Réception d'un datagramme (méthode bloquante). Ce datagramme comprend, en plus des données (et des informations sur ces données : taille, offset), des informations concernant l'hôte distant (adresse IP, numéro de port). */

public void setSendBufferSize(int size);
/* Augmenter la taille du buffer d'envoi (mise à jour de SO_SNDBUF) */

public void setReceiveBufferSize(int size);
/* Augmenter la taille du buffer de réception (mise à jour de SO_RCVBUF) */

public void setSoTimeout (int timeout);
/* Définir un délai max d'attente pour la réception (receive()). Mise à jour de la valeur SO_TIMEOUT ... */

public void connect(InetAddress addrDist, int portDist);
public void connect(SocketAddress addrDist, int timeout);
/* Passer en mode connecté (éventuellement avec un délai d'attente maxi qui permet de ne pas rester bloquer indéfiniment si la socket distante est occupée) */

public void disconnect();
/* Repasser en mode déconnecté */

public void setBroadcast(boolean on);
/* Active/désactive le mode broadcast (met à jour SO_BROADCAST) */

public void setReuseAddress(boolean on);
/* Permet de lier plusieurs sockets sur le même adressage (adresse IP, numéro de port). Mise à jour SO_REUSEADDR. Cette fonctionnalité n'est pas supportée par toutes les sockets (utiliser la méthode public boolean getReuseAddress() pour le savoir). */

public close();
/* Ferme la socket. Les receive(...) en attente de réception léveront une SocketException */

Classe **MulticastSocket** : (hérite de DatagramSocket) : Pour les sockets déconnectées en communication multicast (envoyer et recevoir des datagrammes sur des adresses de groupes)

public MulticastSocket();
public MulticastSocket(int portLoc);
public MulticastSocket(SocketAddress addrLoc);
/* Constructeur de la classe. Création d'une socket multicast (en fixant ou non l'adresse IP et/ou le numéro de port à utiliser). Une fois la socket créée, la méthode DatagramSocket.setReuseAddress(boolean) est appelée pour activer SO_REUSEADDR */

public void setTimeToLive(int ttl);
/* définir une durée de vie (nombre de routeurs traversés, entre 0 et 255) pour les datagrammes multicast envoyés. Les datagrammes avec un ttl de 0 ne peuvent pas être émis sur le réseau, avec un TTL de 1 on ne sort pas du réseau local ... */

public void joinGroup(InetAddress addrMulticast);
/* Rejoindre un groupe multicast. En paramètre, une adresse multicast. */

public void leaveGroup(InetAddress addrMulticast);
/* Quitter le groupe */

public void setInterface(InetAddress inf);
/* Définit une interface d'adressage pour envoyer et recevoir des datagrammes multicast (utile si la machine contient plusieurs cartes réseau pour ne pas laisser le système choisir). */

Classe **DatagramPacket** : Manipulation de datagrammes

DatagramPacket(byte[] buf, int length);
DatagramPacket(byte[] buf, int offset, int length);
DatagramPacket(byte[] buf, int length, InetAddress addrDist, int portDist);
DatagramPacket(byte[] buf, int offset, int length, InetAddress addrDist, int portDist);
DatagramPacket(byte[] buf, int offset, int length, SocketAddress addrDist);
DatagramPacket(byte[] buf, int length, SocketAddress addrDist);
/* Constructeur de la classe. Le paramètre buf contient les données à envoyer, length indique leur taille (qui doit être <= à la taille du buffer) et offset est la position de départ pour la lecture/écriture (décalage par rapport au début du buffer). Il faut utiliser l'un des deux premiers constructeurs pour la réception et l'un les quatre suivants pour l'envoi (addrDist et portDist permettent d'adresser le destinataire). */

public int getOffset();
/* Retourne l'offset des données à envoyer ou reçues */

public int getLength();
/* Retourne leur taille */

public byte[] getData();
/* Récupère les données (soit length octets à partir de la position offset). */

public SocketAddress getSocketAddress();
/* Retourne l'adresse IP et le numéro de port de la socket distante à laquelle ce datagramme est envoyé ou depuis laquelle ce datagramme est reçu */

public InetAddress getAddress();
/* Retourne son adresse IP. */

public int getPort();
/* Retourne son numéro de port. */

Classe abstraite **SocketAdress** : Manipulation d'informations d'adressage (adresse IP, numéro de port)

Classe **InetSocketAdress** : Hérite de la précédente

InetSocketAddress(InetAddress addr, int port);
InetSocketAddress(String host, int port);
/* Constructeurs. Si le numéro de port est à 0, c'est le système qui en choisit un */

public final InetAddress getAddress();
/* Retourne l'adresse IP (de l'objet InetSocketAdress courant) dans un objet InetAdress. */

public final int getPort();
/* Retourne le numéro de port (de l'objet InetSocketAdress courant). */

public final String getHostName();
/* Retourne le nom d'hôte (de l'objet InetSocketAdress courant). */

Classe **InetAdress** : Manipulation d'adresses IP

public byte[] getAddress();
/* Retourne l'adresse IP de l'objet InetAdress courant dans un tableau d'octets (tableau de 4 octets pour IPv4, de 16 octets pour IPv6) */

public String getHostAddress();
/* Retourne l'adresse IP de l'objet InetAdress courant dans une chaîne. */

public String getHostName();
/* Retourne le nom d'hôte de l'objet InetAdress courant dans une chaîne. */

public static InetAddress getByAddress(byte[] addr);
/* A partir d'une adresse IP fournie sous forme d'un tableau d'octets (tableau de 4 octets en IPv4, 16 en IPv6), retourne un objet InetAdress */

public static InetAddress getByName(String host);
/* A partir d'un nom d'hôte (nom complet tel que "java.sun.com" ou adresse IP textuelle), retourne un objet InetAdress. Si null est passé en paramètre, c'est l'adresse de loopback qui est retournée dans l'objet InetAdress. */

public static InetAddress[] getAllByName(String host);
/* A partir d'un nom d'hôte (nom complet ou @IP textuelle), retourne un tableau d'objets InetAdress correspondant aux différentes adresses IP de l'hôte */

public static InetAddress getLocalHost();
/* Retourne l'adresse IP de l'hôte local dans un objet InetAdress */

Questions de cours sur les sockets en C

1) Citez 2-3 fonctions de la couche Socket que l'on peut trouver dans le code d'une communication point à point mais pas dans celui d'une communication multipoint (broadcast ou multicast).

2) Paramètres d'entrée / de sortie

a) Pour chacune des fonctions suivantes, indiquer quels sont les paramètres d'entrée (ceux dont la fonction a besoin), de sortie (ceux qui sont initialisés par la fonction) et d'entrée/sortie (ceux qui mis à jour par la fonction).

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

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

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

*ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);*

*ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr * from, socklen_t *fromlen);*

b) Un paramètre d'entrée peut-il être passé par valeur ? Par adresse ? Et un paramètre de sortie ?

3) Les fonctions d'émission et de réception de message en mode non connecté (*sendto(...)* , *recvfrom(...)*) contiennent dans leurs paramètres les informations d'adressage de la socket distante. Ce n'est pas le cas des fonctions d'envoi et de réception en mode connecté (*send(...)*, *recv(...)*). Pourquoi ?

4) Lorsqu'on utilise la fonction *bind(...)*, a-t-on besoin des informations d'adressage de la socket courante ? De la socket distante ?

5) Parmi les fonctions suivantes, lesquelles sont des "fonctions réseau", c'est à dire émettant ou récupérant un appel réseau. Outre les délais d'émission et réception sur le réseau, y a t-il parmi elles une ou des fonction(s) bloquante(s) ? Expliquer.

a) *socket(...)*

b) *bind(...)*

c) *connect(...)*

d) *send(...)*

e) *accept(...)*

f) *recv(...)*

9) La fonction *accept()* créé-t-elle un nouveau processus ?

Questions de cours sur les sockets en Java

1) Les interfaces, classes et méthodes du package *java.io* sont-elles utiles en mode connecté ? Déconnecté ?

2) On trouve les instructions suivantes dans un programme : Est-ce correct ? Que fait-on dans l'ordre ? De quel côté se situe-on à priori : client / serveur / on ne sait pas ? Comment communique t-on à priori : en mode connecté / déconnecté / on ne sait pas ? Connait-on l'adresse IP et le numéro de port utilisés par la socket courante ?

```
Socket mySock = new Socket();

mySock.connect(new InetSocketAddress("134.59.58.41", 2222));

BufferedReader in = new BufferedReader(new
InputStreamReader(mySock.getInputStream()));
```

3) A quoi sert la méthode *flush()* de la classe *OutputStream* (ou *OutputStreamWriter* ou *BufferedWriter*) puisqu'on l'appelle généralement après avoir écrit dans le flot de sortie ?

4) La classe *MulticastSocket* ne possède pas de méthodes de type *send(...) & receive(...)*. Comment une socket multicast (objet de type *MulticastSocket*) peut-elle envoyer & recevoir des messages (datagrammes) ?