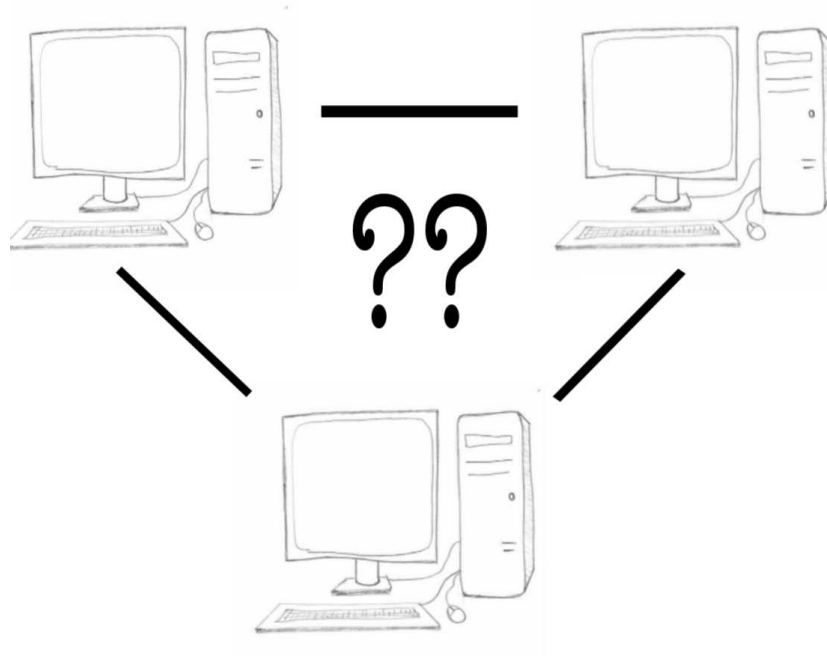


PROGRAMMATION RÉSEAU

Arnaud Sangnier

sangnier@liafa.univ-paris-diderot.fr

API TCP C (1)



Rappel communication par flux

- Dans la communication **par flux** (comme **TCP**)
 - On se connecte (ou on attend une connexion)
 - On échange des messages
 - Les messages arrivent dans l'ordre dans lequel ils sont émis
 - On se déconnecte (ou on ferme une connexion)
 - Pour se connecter il faut :
 - L'adresse d'une machine
 - Le port sur lequel on va se connecter
- On a vu comment faire cela en **Java** aujourd'hui on va voir cela en **C**
 - L'API C est de plus bas niveau
 - Beaucoup plus de choses à faire soi-même

Les adresses internet en C

- **Rappel** : aujourd'hui les adresses internet peuvent être de deux types
 - les adresses **IPv4** sur 4 octets (donc 32 bits)
 - par exemple : **173.194.66.106**
 - les adresses **IPv6** sur 16 octets (donc 128 bits) ; 8 groupes de 2 octets
 - par exemple : **2a00:1450:400c:c02:0:0:0:93**
- Types des variables stockant les adresses IPv4 :
 - **struct in_addr** ou **in_addr_t**
- Types des variables stockant les adresses IPv6 :
 - **struct in6_addr**
- Il n'est pas nécessaire de connaître la structure interne de ces types (i.e. aucune raison d'avoir à manipuler l'intérieur de la structure soi-même)
- Le fichier à inclure pour manipuler ces types :
 - **#include <netinet/in.h>**

Manipulation des structures d'adresse

- Différentes fonctions permettent de manipuler les structures précédentes
- En particulier pour les traduire vers une chaîne de caractères et vice-versa
- Inclure le fichier **<arpa/inet.h>**
 - **char * inet_ntoa(struct in_addr)**
 - Traduit une adresse IPv4 en chaîne caractères
 - **int inet_aton(const char *,struct in_addr *)**
 - Met l'adresse donnée par la chaîne de caractères dans le deuxième argument
 - Renvoie 0 si l'adresse n'est pas valide
 - **Penser à tester les erreurs en C**
 - **in_addr_t inet_addr(const char*)**
 - Similaire à inet_aton

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main() {
    struct in_addr address;
    char *string_address;
    inet_aton("127.0.0.1", &address);
    string_address=inet_ntoa(address);
    printf("L'adresse vaut : %s\n", string_address);
    return 0;
}
```

Manipulation des structures d'adresse

- Pour les adresses IPv6, il existe des fonctions génériques :
 - **const char *inet_ntop(int af, const void *src, char *chaine, socklen_t size)**
 - **const char *inet_pton(int af, char *chaine, void *dest)**
- « presentation to network » and « network to presentation »
- L'entier **af** représente la famille protocolaire (**AF_INET** ou **AF_INET6**)
- **src** et **dst** sont des pointeurs vers des adresses internet conformes à la valeur de **af**
- **socklen_t** est la taille maximale que l'on peut mettre dans chaine
 - Valeur utiles **INET_ADDRSTRLEN** et **INET6_ADDRSTRLEN**
- **ATTENTION** : Ces fonctions ne font PAS appel à l'annuaire. Elles ne font que des transformations entre représentations !!!!

Les sockets en C

- **Rappel** : une socket est un point de communication
- Une socket est caractérisée par :
 - une adresse Internet
 - un numéro de port
 - un type de communication (UDP ou TCP) - bien entendu le type de communication est le même aux deux extrémités de la socket
- En C, une socket est représentée par le type **sockaddr**

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

```
struct sockaddr {  
    sa_family_t sa_family;  
    char      sa_data[];  
};
```

Les sockets en C (2)

- Le champ **sa_family** permet de spécifier le type de la socket (et donc la structure implémentant **sockaddr**). On a les constantes suivantes :
 - **AF_LOCAL** ou **AF_UNIX** pour une socket « locale »
 - **AF_INET** pour une socket IPv4
 - **AF_INET6** pour une socket IPv6
- Nous ne nous intéresserons qu'aux sockets du domaine Internet (**AF_INET** ou **AF_INET6**)
- En fait **struct sockaddr** est une structure générale qui est implémentée par des structures plus spécifiques (en particulier le tableau **char sa_data**)

Structures des adresse de sockets

- Type pour les sockets IPv4

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

```
struct sockaddr_in {
```

```
    short      sin_family; // famille de socket (ex : AF_INET)
```

```
    unsigned short sin_port; // numéro de port : htons(3490)
```

```
    struct in_addr sin_addr; // adresse internet IPv4
```

```
    char      sin_zero[8]; // souvent rempli avec des zéros
```

```
};
```

Structures des adresses de sockets

- Type pour les sockets IPv6

```
struct sockaddr_in6 {  
    u_int16_t    sin6_family; // famille de socket AF_INET6  
    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;  
};
```

Création d'adresses de socket

```
struct sockaddr_in adress_sock;  
  
adress_sock.sin_family = AF_INET;  
adress_sock.sin_port = htons(3490);  
inet_aton("10.0.0.1",&adress_sock.sin_addr);
```

Codage des entiers ?



C'est quoi cette
fonction htons ?

- La représentation des nombres peut être variée, or comme on communique entre machines, il est nécessaire de se mettre d'accord
- Deux grands types de codage des entiers :
 - **petit-boutiste** ou **petit-boutien** (**little-endian**)
 - **grand-boutiste** ou **grand-boutien** (**big-endian**)

Codage des entiers

- Le codage d'un entier n en base b s'écrit de la façon suivante :
 - $n = \sum c_i b^i$
- Comme les entiers utilisent des octets on peut considérer la base b comme valant 256 (c'est à dire 2^8)
- Pour un entier de 32 bits, il faut donc 4 octets et un entier n s'écrit de la façon suivante :

$$n = c_3 \times 256^3 + c_2 \times 256^2 + c_1 \times 256 + c_0$$

- En pratique on stocke dans un tableau de 4 octets les chiffres c_3, c_1, c_2 et c_0
- Ce qui change c'est l'ordre dans lesquels sont stockés dans le tableau ces 4 valeurs

Little-endian vs Big-endian

- Stockage de $n = c3 \times 256^3 + c2 \times 256^2 + c1 \times 256 + c0$
- En **big-endian**

0	1	2	3
c3	c2	c1	c0

- En **little-endian**

0	1	2	3
c0	c1	c2	c3

- **IMPORTANT** : C'est l'ordre des entiers qui changent pas l'ordre des bits dans un entier

Passer d'une machine au réseau

- Sur une machine, l'entier peut-être codé en big-endian ou little-endian (dépend du système)
- Sur le réseau, pour le protocole IP les entiers sont codés en Network Big Order (NBO) qui correspond au big-endian
- Il faut donc convertir les représentations des entiers

```
#include <arpa/inet.h>
```

```
uint32_t htonl(uint32_t hostlong);
```

```
uint16_t htons(uint16_t hostshort);
```

```
uint32_t ntohl(uint32_t netlong);
```

```
uint16_t ntohs(uint16_t netshort);
```

- **h** vaut pour host et **n** pour network

Passer d'une machine au réseau

- Ainsi pour passer le port numéro 3490 sur notre machine et le mettre dans la bonne représentation pour le réseau on fait :
 - **htons(3490)**
 - *host to network short* (short car le port est un entier short)
- **Nota Bene :**
 - En C il faut faire attention de bien faire les conversions pour toutes les données utilisées
 - Si votre machine est en big-endian, la conversion ne fait rien
 - Mais si vous testez le même programme sans conversion sur une autre machine, il pourrait y avoir des problèmes !!!
 - En Java, ce n'est pas nécessaire, l'API fait la conversion elle-même

Accès à une machine

```
inet_aton("10.0.0.1", &adress_sock.sin_addr);
```



Mais si on ne connaît
pas l'adresse IP ?

- Il faut interroger l'annuaire DNS

Interroger l'annuaire en C

- On peut désirer récupérer l'adresse Internet associée à un nom Internet
 - Ceci nécessite d'obtenir **la résolution de nom**
 - Il existe différentes fonction d'accès à l'annuaire DNS
 - La fonction historique
 - **struct hostent *gethostbyname(const char *name);**
 - La fonction moderne
 - int getaddrinfo(const char *node, // "www.example.com" or IP**
const char *service, // "http" or port number
const struct addrinfo *hints,
struct addrinfo **res);

La fonction gethostbyname

- **#include <netdb.h>**

struct hostent *gethostbyname(const char *name);

- L'appel à cette fonction renvoie une structure de la forme suivante :

struct hostent {

char *h_name; // Le nom canonique

char **h_aliases; // Une liste d'alias - le dernier élément est NULL

int h_addrtype; // Le type de l'adresse, qui devrait être AF_INET en général

int h_length; // La longueur des adresses en octet

char **h_addr_list; // Une liste d'adresses IP pour cet host

};

- En fait la dernière est un tableau de **struct in_addr ***, le dernier élément est NULL aussi

Exemple

- On va faire un code qui pour un nom de machine va récupérer toutes les adresses IPv4 correspondantes et les afficher. On affichera également les alias associés à un nom
- Pour cela :
 - On récupère le hostent correspondant
 - On parcourt les tableaux d'alias et d'adresses
 - Pour chaque adresse, on la traduit en chaîne de caractères grâce à la fonction :

char * inet_ntoa(struct in_addr)

Passer d'une machine au réseau

- Sur une machine, l'entier peut-être codé en big-endian ou little-endian (dépend du système)
- Sur le réseau, pour le protocole IP les entiers sont codés en Network Big Order (NBO) qui correspond au big-endian
- Il faut donc convertir les représentations des entiers

```
#include <arpa/inet.h>
```

```
uint32_t htonl(uint32_t hostlong);
```

```
uint16_t htons(uint16_t hostshort);
```

```
uint32_t ntohl(uint32_t netlong);
```

```
uint16_t ntohs(uint16_t netshort);
```

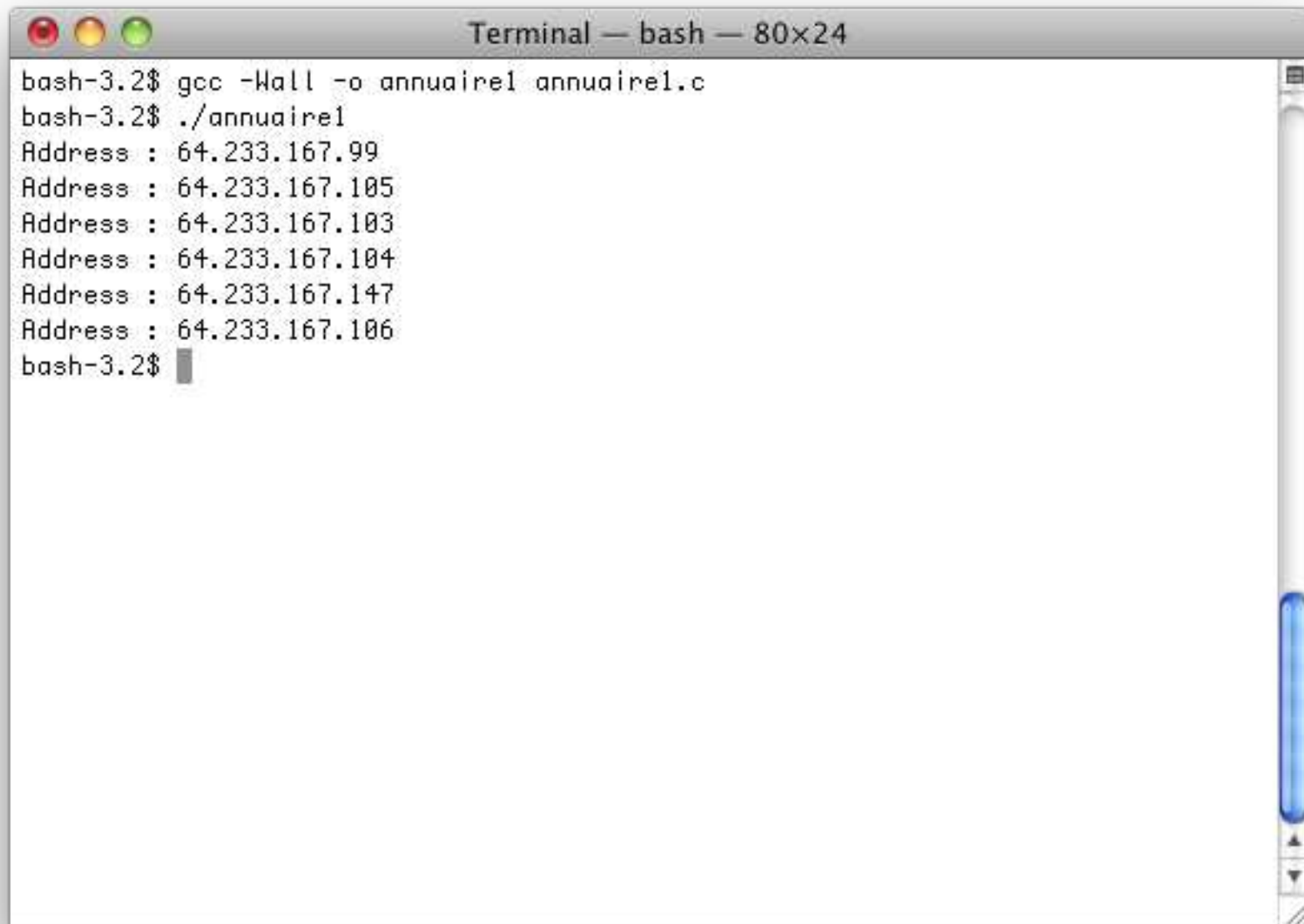
- **h** vaut pour host et **n** pour network

Récupération d'IP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

int main() {
    struct hostent* host;
    host=gethostbyname("www.google.com");
    if(host==NULL) {
        printf("Unknown\n");
    }
    char **aliases=host->h_aliases;
    while(*aliases!=NULL){
        printf("Alias : %s\n",*aliases);
        aliases++;
    }
    struct in_addr **addresses=(struct in_addr**)host->h_addr_list;
    while(*addresses!=NULL){
        printf("Address : %s\n",inet_ntoa(**addresses));
        addresses++;
    }
    return 0;
}
```

Résultat

A screenshot of a macOS Terminal window titled "Terminal — bash — 80x24". The window shows the execution of a C program named "annuaire1.c". The user enters the command "gcc -Wall -o annuaire1 annuaire1.c" to compile the program, followed by "./annuaire1" to run it. The program outputs six IP addresses, each preceded by "Address :". The addresses are 64.233.167.99, 64.233.167.105, 64.233.167.103, 64.233.167.104, 64.233.167.147, and 64.233.167.106. The prompt "bash-3.2\$" is visible at the end of each line.

```
bash-3.2$ gcc -Wall -o annuaire1 annuaire1.c
bash-3.2$ ./annuaire1
Address : 64.233.167.99
Address : 64.233.167.105
Address : 64.233.167.103
Address : 64.233.167.104
Address : 64.233.167.147
Address : 64.233.167.106
bash-3.2$
```

La fonction getaddrinfo

- Cette fonction est plus générique mais donc plus complexe à utiliser !!!
- C'est la fonction que l'on recommande d'utiliser

```
int getaddrinfo(const char *node,  // "www.example.com" or IP
               const char *service, // "http" or port number
               const struct addrinfo *hints,
               struct addrinfo **res);
```

- On ne décrira que partiellement son utilisation
- Cette fonction permet d'obtenir entre autres choses une liste d'adresses (au sens très large) associées à un nom Internet dans l'annuaire
- En pratique elle remplit une structure de type **struct addrinfo** qui est stockée dans la variable **res**
- On remarque qu'on peut donner aussi un numéro de port (mais on peut mettre NULL, si on veut juste un adresse)
- Cette fonction renvoie 0 si tout se passe bien

La structure struct addrinfo

```
struct addrinfo {  
    int  ai_flags;  
    int  ai_family; // la famille du protocole AF_xxxx  
    int  ai_socktype; // le type de la socket SOCK_xxx  
    int  ai_protocol;  
    socklen_t  ai_addrlen; // la longueur de ai_addr  
    struct sockaddr *ai_addr; // l'adresse binaire  
    char  *ai_canonname; // le nom canonique  
    struct addrinfo *ai_next; // le pointeur vers la structure suivante  
};
```

- Il s'agit d'une liste chaînée, **ai_next** est le successeur
- Il faut libérer la mémoire de la liste après utilisation grâce à

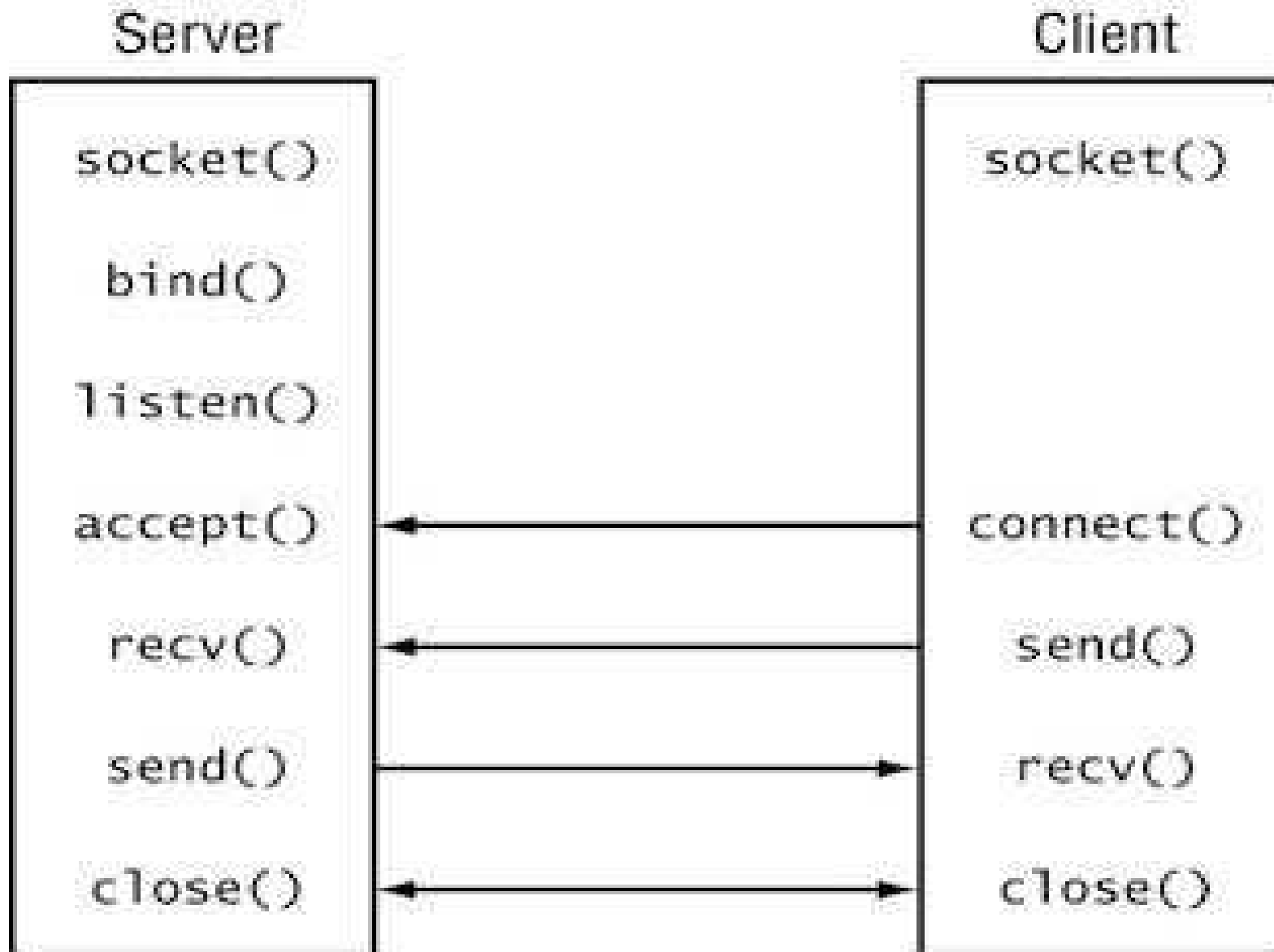
```
void freeaddrinfo(struct addrinfo *);
```

Récupération d'IP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

int main() {
    struct addrinfo *first_info;
    int r=getaddrinfo("www.google.com",NULL,NULL,&first_info);
    if(r==0){
        struct addrinfo *info=first_info;
        while(info!=NULL){
            struct sockaddr *saddr=info->ai_addr;
            if(saddr->sa_family==AF_INET){
                struct sockaddr_in *addressin=(struct sockaddr_in *)saddr;
                struct in_addr address=(struct in_addr) (addressin->sin_addr);
                printf("Address : %s\n",inet_ntoa(address));
            }
            info=info->ai_next;
        }
    }
    freeaddrinfo(first_info);
    return 0;
}
```

Schéma Client-Serveur en C



Création d'une socket

- La création d'une socket se fait grâce à :
 - **#include <sys/socket.h >**
 - int socket(int domaine, int type, int protocol)**
- Pour nous :
 - **domaine** vaudra **PF_INET** (pour IPv4) ou **PF_INET6** (pour IPv6)
 - **type** vaudra **SOCK_STREAM** (pour les sockets TCP)
 - **protocol** spécifie le protocole de communication (mais pour TCP, on peut mettre 0 et le protocole est choisi de façon automatique)
- L'entier renvoyé sera le descripteur utilisé pour communiquer

Accès à une machine

int socket(int domaine, int type, int protocol)



Mais là on dit jamais
l'adresse ou le port

- Et oui !!! Ce n'est pas comme en java ! Il faut associer la socket à un point de communication

Côté client

- Il faut demander l'établissement d'une connexion à l'aide de la fonction suivante :

int connect(int socket, const struct sockaddr *adresse, socklen_t longueur);

- On connecte la socket correspondante
- Pour rappel dans les objets de type struct sockaddr_in, on met une adresse et un port
- Pour le dernier argument, si on est en IPv4 et que adresse est de type **struct sockaddr_in**, on pourra mettre **sizeof(struct sockaddr_in)**
- Quand on a fini la communication, on peut fermer le descripteur de socket avec la commande
 - **int close(int fildes);**

Pour communiquer

- On va envoyer et recevoir des caractères sur le descripteur de socket
- Pour recevoir on va utiliser
- **int recv(int sockfd, void *buf, int len, int flags);**
 - Remplit le buffer **buf**
 - **len** est la taille maximale de **buf**
 - **flags** sera la plupart du temps mis à **0**
 - renvoie le nombre de données reçu (-1 si erreur et 0 si la connexion est fermée)
- Pour envoyer on va utiliser
- **int send(int sockfd, const void *msg, int len, int flags);**
 - Même principe que recv len est la taille en octet de msg
 - flags est aussi mis à 0 ici.
- On pourrait aussi utiliser **read** et **write**

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>

int main() {
    struct sockaddr_in address_sock;
    address_sock.sin_family = AF_INET;
    address_sock.sin_port = htons(4242);
    inet_aton("127.0.0.1",&address_sock.sin_addr);

    int descr=socket(PF_INET,SOCK_STREAM,0);
    int r=connect(descr,(struct sockaddr *)&address_sock,
        sizeof(struct sockaddr_in));

    if(r!=-1){
        char buff[100];
        int size_rec=recv(descr,buff,99*sizeof(char),0);
        buff[size_rec]='\0';
        printf("Caracteres recus : %d\n",size_rec);
        printf("Message : %s\n",buff);
        char *mess="SALUT!\n";
        send(descr,mess,strlen(mess),0);
    }
    return 0;
}
```