**Synthèse RTI janvier 2024**

1 - Communications réseaux en C :

( Appels systems : socket(), bind(), listen(), accept(), connect(), read(), write(), close() )

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

• Domain est le « domaine de communication » → il définit une famille d’adresses

(plus exactement le format de adresses possibles). Il peut prendre les valeurs

suivantes :

o AF\_INET : protocole fondé sur IPv4

o AF\_INET6 : protocole IPv6 (soumis à des options de compilation

particulières)

o AF\_UNIX : communication limitée aux processus résidant sur la même

machine

o …

• Type est le type de socket → mode connecté (TCP) ou mode non connecté (UDP). Il

peut prendre les valeurs suivantes :

o SOCK\_STREAM : mode connecté (TCP)

o SOCK\_DGRAM : mode non connecté (UDP)

o SOCK\_RAW : pour dialoguer de manière brute avec le protocole (le plus

souvent IP)

• Protocol est le protocole désiré. Il peut prendre les valeurs

o IPPROTO\_TCP

o IPPROTO\_UDP

o IPPROTO\_IP

o …

En ce qui nous concerne, le domaine sera AF\_INET et le protocole sera soit

IPPROTO\_TCP, soit IPPROTO\_UDP. Dans ce cas, le choix du protocole est implicite et le

paramètre protocol peut être mis à 0.

**Le retour de l’appel système est**

• Le descripteur de la socket (un entier supérieur ou égale à 0) si tout s’est bien passé

• -1 en cas d’erreur (et errno est positionné)

**int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);**

• **sockfd** est le descripteur de la socket que l’on désire lier

• **addr** est un pointeur vers une structure contenant l’adresse IP et le port souhaité

→ il s’agit en fait ici d’un pointeur générique → il pointera vers une structure

spécifique correspondant à la famille d’adresse utilisée (AF\_INET, AF\_INET6, …)

• **addrlen** est la taille (en octets) de la structure pointée par addr

Dans le cas présent (mode connecté – TCP – domaine AF\_INET), l’adresse utilisée sera

décrite à l’aide de la structure sockaddr\_in :

struct sockaddr\_in {

sa\_family\_t sin\_family; // Famille d'adresses (AF\_INET)

in\_port\_t sin\_port; // Numéro de port

struct in\_addr sin\_addr; // Adresse IP

char sin\_zero[8]; // Remplissage pour alignement

};

La structure in\_addr représente donc une adresse IP et est définie par

struct in\_addr {

in\_addr\_t s\_addr; // Adresse IP en format réseau

};

**int getaddrinfo(const char \*host, const char \*service, const struct addrinfo \*hints, struct addrinfo \*\*result);**

• **host** est le nom de la machine hôte (exemples : « www.google.be » , « moon », …) ou une adresse IP (exemple : « 142.251.36.3 », …) fourni sous la forme d’une chaîne de caractères

• **service** est le service (exemple : « http ») ou le numéro de port (exemples : « 80 », « 50000 », …) fourni sous la forme d’une chaîne de caractères

• **hints** est une structure contenant les détails de notre recherche

• **result** est le résultat de la recherche (le résultat de la fonction) fourni sous la forme d’une liste chainée de structures addrinfo

La structure addrinfo est définie par :

struct addrinfo {

int ai\_flags;

int ai\_family;

int ai\_socktype;

int ai\_protocol;

**socklen\_t ai\_addrlen;**

**struct sockaddr \*ai\_addr;**

char \*ai\_canonname;

struct addrinfo \*ai\_next;

};

• **ai\_family** est la famille d’adresses souhaitées (par exemple AF\_INET)

• **ai\_socktype** est le type de socket souhaitée (par exemple SOCK\_STREAM ou SOCK\_DGRAM)

• **ai\_protocol** est le protocole utilisé (par exemple IPPROTO\_TCP ou IPPROTO\_UDP, on peut mettre 0 s’il n’y a pas d’ambiguïté possible)

• **ai\_flags** contient un ou des flags (combinés par |) :

o AI\_NUMERICHOST : si l’argument host contient une adresse IP numérique plutôt qu’un nom d’hôte

o AI\_NUMERICSERV : si l’argument service contient un numéro de port et non pas un nom de service

o AI\_PASSIVE : si l’adresse demandée sera utilisée pour y attacher un serveur avec bind()

o …

Lors du retour de la fonction, le pointeur **result** est initialisé au début d’une liste chainée

de toutes les structures **addrinfo** correspondant à notre demande. La liste est chainée à

l’aide du champ ai\_next.

Lors de l’appel de cette fonction, host ou service peut être NULL mais pas les deux

simultanément.

Les champs remplis par l’appel de la fonction sont

• **ai\_addr** est un pointeur sur la structure d’adresse convoitée (ce que l’on passera

au second argument de l’appel système bind())

• **ai\_addrlen** est la taille de cette structure (ce que l’on passera en 3ème paramètre de

l’appel système bind()).

**int listen(int sockfd, int count);**

• **sockfd** est la socket sur laquelle on souhaite attendre des connexions

• **count** est un paramètre qui spécifie le nombre maximum de connexions qui

peuvent être reçues par le serveur mais qui n’ont pas encore été prises en compte

par celui-ci au moyen de l’appel système accept() (voir plus loin) : de telles

demandes sont appelées connexions pendantes et sont enfilées dans une FIFO. La

taille maximum de cette FIFO est fixée par la constante SOMAXCONN

L’appel de cette fonction n’est pas bloquant et retourne

• 0 en cas succès

• -1 en cas d’erreur et ERRNO est positionné. Une valeur particulièrement

intéressante de ERRNO est EADDRINUSE qui signifie qu’une autre socket est déjà à

l’écoute sur le même port

**int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);**

• **sockfd** est la socket sur laquelle on désire attendre une connexion

• **addr** est un pointeur vers une structure d’adresse réseau correspondant à celle du

client qui vient de se connecter sur le serveur (on parle de « peer socket ») →

celle-ci sera remplie par la fonction au moment de l’acception de la connexion par

le serveur

• **addrlen** est la taille de cette structure → elle est passée par adresse à la fonction

car elle va être initialisée par la fonction

Cette fonction est bloquante (sauf s’il y a des connexions pendantes) jusqu’au moment

où un client se connecte sur le serveur (voir plus loin) et retourne

• -1 en cas d’erreur, mais surtout

• Un entier positif correspondant à une socket dupliquée → ceci a pour effet

d’ouvrir une nouvelle socket côté serveur, appelée socket de service, qui est mise

en connexion avec le client. La socket originale, appelée socket d’écoute, est

restée intacte et est prête à servir à nouveau pour une demande de connexion

(listen() + accept())

**int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);**

• **sockfd** est une socket créée grâce à l’appel de socket() et qui sera associée à la

future connexion avec notre processus serveur

• **addr** est l’adresse d’une structure contenant l’adresse réseau du processus serveur

que l’on désire contacter

• **addrlen** est la taille de cette structure

Si tout s’est bien passé,

• le retour de la fonction est 0

• si aucun bind() n’a été réalisé par le processus appelant (notre client), le système

attribue automatiquement un port local à la socket désignée par sockfd

• la communication est établie avec le processus serveur dont l’appel système

accept() s’est débloqué (voir plus haut)

En cas d’erreur la fonction retourne -1.

**ssize\_t write(int fd, const void \*buf, size\_t count);**

**ssize\_t read(int fd, void \*buf, size\_t count);**

• **fd** est le descripteur de la socket de service du client ou du serveur

• **buf** a le même rôle que dans les appels système send() et recv()

• **count** à le même rôle que len dans les appels système send() et recv()

**int close(int fd);**

où fd est la socket que l’on désire fermer.

Si cela n’est pas fait, les sockets de service risquent de rester dans des états comme

CLOSE\_WAIT, FIN\_WAIT1, … Et il faudra alors attendre un certain temps que le système

libère correctement le port avant de pouvoir le réutiliser.

Evidemment, il sera également nécessaire de fermer la socket d’écoute du serveur

lorsqu’il ne souhaitera plus accepter de nouvelles connexions.

2 - Les threads en Java :

classe Thread, interface Runnable, méthodes synchronized, ThreadGroup

3 - JDBC et l’accès aux bases de données :

les classes Class (et sa méthode forName()), DriverManager, les interfaces Connection, Statement, ResultSet

4 - Communications réseaux en Java :

classes InetAddress, ServerSocket, Socket

5 - Cryptographie et Java :

les interfaces Key, Cipher, SecretKey, PublicKey, PrivateKey, les classes Provider, KeyGenerator, KeyPairGenerator, KeyPair, MessageDigest, Mac, Signature, Certificate, X509Certificate, KeyStore

6 - Programmation Réseau Web :

classes Java HttpServeur, HttpContext, HttpExchange, l’interface Java HttpHandler. Classes Javascript XMLHttpRequest et méthodes relatives au DOM