Communication réseau (TCP-IP)

A. La communication client-serveur

A.1. Le concept client-serveur

TCP-IP fournit un ensemble de protocoles pour la communication entre deux processus. Ces protocoles définissent principalement les formats des trames à échanger. Aucune contrainte n'est imposée quant à la méthodologie à employer pour créer des applications en réseau. Le concept utilisé pour formaliser la communication entre deux processus distants est appelé client-serveur.

A.1.1 Le modèle client-serveur

Le principe du client-serveur part de l'observation suivante : quand deux tâches ont besoin de communiquer, il faut qu'elles soient actives toutes les deux sur leurs machines respectives. En effet TCP-IP n'offre aucun mécanisme permettant de lancer une tâche sur une machine distante. Il faut donc laisser la tâche qui offre un service en permanence à l'écoute des clients.

La tâche qui offre (sert**) un service est située sur le serveur.**

La tâche qui demande le service est située sur un client.

Deux cas se présentent au moment du réveil de la tâche

- la tâche traite la requête du client, renvoie le résultat et se remet en sommeil attendant une nouvelle requête client.
- la tâche se duplique (fork()) permettant à son fils de répondre à la requête du client tandis qu'elle (la mère) se remet à l'écoute des clients éventuels. C'est la solution retenue quand le traitement de la requête est long. La tâche serveur peut alors répondre à plusieurs requêtes en même temps. Le serveur est dit à **accès concurrents**.

A.1.2 Principe résumé du client-serveur

Le serveur: C'est une application située sur une machine (aussi appelée serveur), elle est disponible pour offrir un service (fournir des données d'une base de données, réaliser un affichage graphique, ...).Elle est à l'écoute des requêtes qui lui parviennent. En principe elle ne prend pas l'initiative d'une communication. Elle ne connaît pas l'adresse de ses clients tant que la communication n'est pas établie. Elle ne dispose de l'adresse d'un client qu'après avoir reçu une requête de celui-ci. L'adresse IP et le numéro de port du serveur sont connus de tous les clients potentiels.

Le client: C'est une application située sur une machine distante du serveur. Elle prend l'initiative d'ouvrir une session de communication avec l'application serveur située sur une machine distante. Elle peut envoyer une demande de connexion (mode connecté) ou directement une requête (mode non-connecté). Elle connaît donc l'adresse IP et le numéro de port de l'application serveur. Elle connaît aussi les services proposés par l'application serveur. L'adresse du client importe peu car personne ne le contacte tant que lui-même n'a pas pris l'initiative d'un contact au cours duquel il transmet ses coordonnées.(@IP et port).

Quand il s'adresse au serveur le client parle le même langage que lui. Il connaît les requêtes que le serveur peut traiter, et il sait sous quelle forme la réponse sera fournie. Après avoir reçu et traité la réponse le client continue son exécution.

A.2. Choix d'un service avec ou sans connexion

Le choix du protocole pour développer le serveur découle directement du choix du type de service. On choisira TCP pour un service avec connexion et UDP pour un service sans connexion.
Comment choisir? (voir aussi B3)

On choisira le protocole UDP dans les trois cas suivants:

- la requête du client et la réponse du serveur sont très courtes (quelques octets).

Exemple: serveur de date: service daytime numéro de port: 13

- l'application est très spécifique et ne peut supporter ni le temps perdu ni la charge réseau imposés par l'établissement d'une connexion.

- l'application prend elle-même en charge les services rendus par TCP (détection et reprise d'erreur, contrôle de flux...)

Dans tous les autres cas on utilisera TCP.

A.3. Applications réparties

Dans une application répartie développée sur le modèle client-serveur, les processus (tâches) peuvent être séparés en deux catégories, d'un côté des processus serveurs et de l'autre des processus clients.

Au niveau serveur il est nécessaire de distinguer:

- la machine sur laquelle un service peut être réalisé (machine serveur),

du processus susceptible de rendre ce service (processus ou tâche serveur)

Au niveau du client, il faut distinguer:

- la machine faisant appel à la machine serveur (machine cliente)
- le processus sollicitant le processus serveur (processus ou tâche cliente)

B. Les primitives de gestion des sockets

B.1. Définition d'une socket

Une socket est un point de communication par lequel un processus peut émettre ou recevoir des informations. A l'intérieur d'un processus, une socket sera identifiée par un descripteur de même nature que ceux identifiant les fichiers. Cette propriété est essentielle car elle permet par exemple la redirection des fichiers d'entrée/sorties standard (descripteurs 9, 1 et 2) sur des sockets et donc l'utilisation sur le réseau d'applications classiques. Tout nouveau processus hérite des descripteurs de sockets de son père. Les sockets peuvent être utilisées par deux processus, de machine différente, pour communiquer à travers le réseau (famille INET) ou par deux processus, d'une même machine, pour échanger des données (famille UNIX).

B.2. Domaine d'une socket

Le domaine d'une socket définit le format des adresses qui pourront être données à la socket. Il définit également une famille de protocoles utilisables.
La structure sockaddr décrite ci-dessous est générique et dans un domaine particulier il faut lui substituer la structure correspondante du domaine.

```
struct sockaddr {
                u_short     sa_family;
                char        sa_data[141;                    } ;
```

B.2.1. Domaine UNIX (AF UNIX)

Dans le domaine AF_UNIX les sockets sont locales au système où elles sont définies. Elles permettent la communication interne entre processus(donc sur la même machine). On parle alors de socket UNIX.
La structure d'une adresse dans le domaine AF UNIX est sockaddr_un qui est définie dans <sys/un.h>.

```
struct sockaddr_un {
                short sun_family;       /* domaine AF_UNIX *l
                char sun_path[1081;   /* référence */
        } ;
```

B.2.2. Domaine INTERNET (AF_INET)

La structure d'une adresse dans le domaine AF_INET est sockaddr_in qui est définie dans <sys/un.h>.

```
struct sockaddr_in {
        short       sin_family;     /* AF_INET *l
        u short     sin_port;       /* Numéro de port sur 2 octets */
        struct in_addr  sin_addr;   /* Adresse internet sur 4 octets*l
        char        sin_zero[8];};  /* Caractères de remplissage à 0*l
```

La structure in_addr est définie comme suit:

```
        struct in_addr { u long     s_addr;     };      /* adresse IP sur 4 octets */
```

B.3. Type de socket

B.3.1 Le type SOCK_DGRAM

Une socket de ce type est orientée vers la transmission de datagrammes.
La communication s'effectue en **mode non connecté**. Le service rendu n'est pas fiable.
Dans le domaine INTERNET le protocole sous-jacent est **UDP**.

B.3.2 Le type SOCK STREAM

Une socket de ce type est orientée vers l'échange de séquences continues de caractères. La communication s'effectue en **mode connecté**. II y a préservation de l'ordre des données. Le service rendu est fiable. Dans le domaine INTERNET le protocole sous-jacent est **TCP**.

B.4. Les primitives principales de l'interface socket

B.4.1 Création d'une socket

La primitive socket() crée un point de communication en créant un descripteur de socket vide.

int socket(int Domaine, int Type, int Protocole);

Domaine: Famille d'adressage, (AF_INET ou AF_UNIX)
Type: SOCK_DGRAM ou SOCK_STREAM
Protocole: La valeur 0 permet de choisir le protocole par défaut.
II n'existe généralement qu'un protocole pour un type et une famille d'adresse donnés.
Exemple: Pour AF_INET et SOCK_DGRAM => protocole UDP
Pour AF INET et SOCK_STREAM => protocole TCP
Cette primitive renvoie le numéro du descripteur de socket ou -1 en cas d'échec.

B.4.2 Attachement côté serveur: bind()

Rappel: Pour la famille d'adressage AF_INET un point de communication est défini par l'association d'une adresse IP et d'un numéro de port. L'adresse IP permet d'identifier la machine hôte, le numéro de port permet de distinguer la tâche qui désire communiquer par ce point de communication. Après l'appel de la primitive socket le point de communication est créé, pour le serveur il faut maintenant lui associer une adresse locale et un numéro de port grâce à la primitive bind().

int bind(int Sock, struct sockaddr *PtrAdresse, int Taille);

Sock est le numéro du descripteur de socket retourné par socket();
Taille est la taille d'une structure de modèle sockaddr.
PtrAdresse contient l'adresse d'une structure contenant tous les paramètres d'initialisation du descripteur.

En cas d'adresse IP invalide, ou de demande d'attachement à un port déjà occupé, bind échoue et renvoie -1 sinon il renvoie 0.

L'attachement est explicite en choisissant le numéro de port. C'est en connaissant ce numéro de port que les clients pourront le contacter. Le numéro de port doit être choisi supérieur à IP_PORT_USER_RESERVED(1023).

Problème de la conversion des entiers

Lors de l'attachement, des entiers sont affectés aux différents champs du descripteur de socket.

La représentation des entiers peut varier d'une machine à l'autre. Certaines utilisent la représentation ***little endian*** (poids faible puis poids fort) et d'autres utilisent la représentation ***big endian*** (poids fort puis poids faible). Les protocoles TCP-IP travaillent en ***big endian***. La solution consiste à utiliser systématiquement des macros de conversions :

Macro	conversion réalisée
htonl(x)	entier long, représentation hôte =>représentation réseau
htons(x)	entier court, représentation hôte =>représentation réseau
ntohl x)	entier long, représentation réseau =>représentation hôte
ntohs(x	entier court, représentation réseau =>représentation hôte

B.4.3. Récupération d'informations concernant le serveur

B.4.3.1. Comment obtenir l'adresse IP d'une machine distante?

Après l'exécution des primitives socket() et bind(), nous disposons d'un point de communication rattaché à la machine locale. Il faut maintenant le relier à un point de communication identique sur la machine distante. Il est nécessaire pour cela de connaître le numéro IP de la machine distante. Il est plus facile d'utiliser ici le nom symbolique de la machine distante. Exemple: saphyr au lieu de "90.9.0.1" Pour cela il est nécessaire, sous UNIX, de consulter le fichier */etc/hosts* qui contient une table associant nom symbolique et numéro IP de toutes les machines du réseau.
La fonction gethostbyname réalise ce travail.

struct hostent *gethostbyname(char *Nom);
Nom pointeur sur le nom de la machine distante

La structure hostent est défini dans <netdb.h>

```
struct hostent {
        char    *h_name;        /* nom de l'hôte *l
        char    **h_aliases     /* liste d'alias */
        int     h_addrtype;     /* AF_INET */
        int     h_length;       /* longueur de l'adresse */
        char    **h_addr_list;  /* liste des adresses de l'hôte *l
        };
```

Sachant que la variable symbolique h_addr est définie par
#define h_addr h_addr_list[0]
L'adresse IP principale de l'hôte est alors accessible par **gethostbyname()->h_addr**

B.4.3.2. Comment obtenir le numéro de port d'un service donné?

Le fichier *etc/services* contient les numéros de port des services disponibles. La fonction getservbyname utilise ce fichier pour renvoyer des informations concernant un service donné.

struct servent *getservbyname(char *Service, char *Protocole);

Service est le nom du service dont on veut obtenir le port.(Exemple telnet, ftp,...).
Protocole est le nom du protocole qu'utilise ce service (certains services sont fournis avec plusieurs protocoles).

```
struct servent {
        char    *s_name;        /* nom officiel du service *l
        char    **s_aliases;    /* liste des alias du service *l
        int     s_port;         /* numéro de port */
        char    *s_proto;   };  l* protocole associé *l
```

On récupère le numéro du port du service par: getservbyname()->s_port

B.4.4. Le point de vue du serveur TCP

B.4.4.1. Mettre un serveur à l'écoute des demandes de connexion: primitive listen();

Cette primitive est utilisée par les serveurs orientés connexion. Cette fonction n'est valable que sur des sockets du type SOCK_STREAM. Après réalisation d'un appel à *listen*, l'application serveur commence à accepter les demandes de connexion en provenance de clients. Si un serveur est très sollicité une demande de connexion peut arriver pendant qu'il traite la demande précédente. Le second paramètre de la primitive permet d'indiquer combien de demandes de connexion seront enregistrées. Une demande de connexion qui est enregistrée mais pas encore servie est dite pendante.

int listen(int Sock, int NbReq);

Sock est le numéro du descripteur de la socket
NbReq nombre de requêtes simultanées que le système accepte de traiter(< 10)

B.4.4.2. Traitement des requêtes en provenance des clients: primitive accept()

Cette primitive est également utilisée par les serveurs orientés connexion .

accept permet à un processus de prendre connaissance de l'existence d'une nouvelle connexion. Cette connexion est extraite de la liste des connexions pendantes et un descripteur sur une socket de service dédiée à cette nouvelle connexion est renvoyé. La communication avec le client a lieu sur la socket de service.

int accept(int Sock, struct sockaddr *PtrAdresse, int *PtrTaille);

Sock est le numéro du descripteur de la socket d'écoute
PtrAdresse contient l'adresse de la socket cliente au retour de la fonction
PtrTaille pointe sur la taille de l'adresse de la socket cliente. PtrTaille est une donnée qu'il faut initialiser.

accept renvoie le numéro du descripteur de la socket de service ou -1 en cas d'erreur. errno contient alors la nature de l'erreur.
accept est une primitive **bloquante** le processus appelant est bloqué s'il n'y a aucune connexion pendante. Une primitive est dite bloquante si elle place la tâche qui l'exécute en attente (ici en attente d'une connexion).
accept réalise donc les opérations suivantes:
- elle traite la première connexion pendante sur la socket en écoute
- une fois la connexion établie, elle crée et attache une nouvelle socket (appelée socket de service ou socket de traitement) qui va traiter la requête du client.

Après acceptation d'une connexion, le serveur a deux possibilités:
- il peut prendre en charge lui même le traitement de la requête du client sur la socket de service (c'est-à-dire rendre le service attendu par le client). L'inconvénient de cette solution est que d'éventuelles connexions ne seront effectivement prises en compte qu'ultérieurement. Il y a risque de saturation des connexions pendantes.
- il peut sous-traiter la requête du client à une tâche fille (lancement de la tâche de traitement) et se remettre immédiatement en attente de connexion. En sous traitant ainsi le traitement d'une requête il est ainsi possible de traiter simultanément plusieurs demandes de connexion, plusieurs tâches de traitement sont alors en exécution simultanées.

B.4.5. Le point de vue du client TCP: la primitive connect()

Cette primitive est utilisée uniquement par un client. Elle est généralement utilisée en mode connecté (protocole TCP). En mode connecté *connect()* réalise entre client et serveur un échange du type poignée de main en 3 étapes permettant l'établissement d'une connexion. Si un appel à la primitive bind n'a pas été fait avant la demande de connexion, un attachement sur un port quelconque sera automatiquement réalisé à l'appel de connect.

int connect(int Sock, struct sockaddr *PtrAdresse, int Taille);

Sock est le numéro du descripteur de la socket locale (qui est cliente).
PtrAdresse contient l'adresse d'attachement du point de connexion distant.
Taille est la taille de la structure pointée par PtrAdresse

Le descripteur est complet, il est possible d'envoyer ou de recevoir des messages sur cette socket. La primitive connect renvoie 0 si la connexion est établie. Elle renvoie -1 en cas d'échec et la variable errno indique la nature de l'erreur.

La connexion est établie si :
- les paramètres sont localement corrects
- il existe une socket de type SOCK_STREAM attachée à l'adresse pointée par PtrAdresse et cette socket est à l'état d'écoute (c'est à dire qu'un appel à la primitive listen a été réalisée sur la socket distante)
- la socket locale n'est pas déjà connectée
- la socket distante n'est pas utilisée dans une autre connexion
- la file des connexions pendantes de la socket distante n'est pas pleine.

B.4.6. Envoyer des messages sur une socket Ces primitives peuvent être utilisées aussi bien sur un serveur que sur un client. Elles sont bloquantes.

Primitive send()

int send(int Sock, char *Message, int TailMessage, int Drapeau);
Sock est le numéro du descripteur de socket
Message pointe sur le message à envoyer
TailMessage indique le nombre d'octets du message
Drapeau option (=0)

send renvoie le nombre de caractères transmis. Si le tampon d'émission est plein le processus appelant est bloqué.

Primitive sendto() Elle n'est utile que dans le **mode non connecté** et si l'on a pas fait appel à la primitive connect(). Elle permet de terminer l'initialisation du descripteur de socket avant d'envoyer un message.
int sendto(int Sock, char *Message, int TailMessage, int Drapeau, struct sockaddr *PtrAdresse, int Taille);
Les quatre premiers arguments sont les mêmes que pour send PtrAdresse contient l'adresse d'attachement du destinataire Taille est la taille de la structure pointée par PtrAdresse

B.4.7. Extraire des messages d'une socket On peut utiliser read pour lire sur un descripteur de socket, mais on préférera les primitives bloquantes suivantes:
int recv(int Sock, char *Message, int TailMessage, int Drapeau);
Arguments identique à send(); recv renvoie le nombre de caractères lus, si le tampon de réception est vide et que le nombre de caractères attendus n'a pas été lus le processus appelant est bloqué.

int recvfrom(int Sock, char *Message, int TailMessage, int Drapeau, struct sockaddr *PtrAdresse, int *PtrTaille); Même arguments que sento mais l'adresse d'attachement est celle de l'expéditeur. PtrTaille pointe sur la taille de cette structure. PtrTaille est une donnée qu'il faut initialiser (Ne pas oublier). Au retour de la fonction, PtrAdresse et PtrTaille permettent de connaître l'expéditeur pour lui répondre.

B.4.8. Fermeture d'une socket : primitive shutdown() ou closesocket( )

int shutdown(int Sock, int Nb);

Nb peut prendre les valeurs suivantes:
- 0 la socket ne peut plus recevoir de données
- 1 la socket ne peut plus envoyer de données (socket passive, écoute seulement)
- 2 fin de la communication (dans les deux sens)