Le modèle Client / Serveur
Les sockets Posix

1 Besoin et principes de fonctionnement :

Dès qu'il y a eu deux ordinateurs sur terre il est apparu nécessaire qu'ils échangent des données. Le modèle Client-Serveur actuellement très répandu répond à ce besoin.

Un processus[1] sur une machine joue le rôle du Serveur, et un processus sur l'autre machine joue le rôle du Client.

Le Serveur attend que le client se connecte. La connexion est donc à l'initiative du client.
Tout processus placé en attente de connexion peut donc à juste titre être appelé « serveur ».

Ce qui se passe ensuite, une fois la connexion établie, est un échange normalisé. Chaque processus sait s'il doit attendre une donnée ou s'il doit en envoyer. Cette synchronisation des échanges peut s'appeler « protocole de dialogue » ou « modalité d'échange ». Elle est définie par le développeur ou par une norme dans le cas d'un protocole standard (http, ftp, ssh, pop, imap, daytime, …).

Sur des systèmes d'exploitation multitâches, chaque machine peut utiliser simultanément les processus client et serveur.
De même, plusieurs machines peuvent être les clientes d'un même serveur.

Bien que l'expression Client-Serveur appartienne largement au monde des réseaux (TCP/IP en autres), on peut utiliser cette expression dès que deux machines communiquent, quelque soit le lien entre les deux machines (ex : liaison série RS232, 485 …).

2 Socket : Le Point de connexion

On appelle « point de connexion » le canal logiciel utilisé par le processus client ou serveur.
Socket est le nom donné au point de connexion du monde TCP/IP et Unix (Communication entre deux processus sur une même machine).
Le projet a été développé à l'origine par l'université de Berkeley, mais il est piloté maintenant par le groupe POSIX, chargé de veiller à la cohérence des différentes distributions Linux. Les deux standards (Posix et Berkeley) cohabitent toujours. Les bibliothèques logicielles sont différentes mais les fonctions utilisées sont quasiment identique.

Depuis, avec l'évolution des systèmes d'exploitation, les sockets ont été portés dans tous les environnements, et dans beaucoup de langages (C, C++, PhP, python, …). Certains environnements (Qt, .NET, …) offrent des bibliothèques plus évoluées permettant d'accéder au réseau via des objets qui éloignent un peu le programmeur de l'aspect « bas niveau » des sockets. Ce n'est souvent qu'un « emballage » des sockets…

3 Le socket Posix et langage C

A l'intérieur du processus, le socket sera identifié par un **descripteur** d'entrées-sorties, ce qui permet l'échange de données avec les primitives classiques d'entrée/sortie : read, write, sendto, recvfrom, send, recv, … comme pour un fichier.

Les processus qui échangent leurs données sont soit sur la même machine (Ex : socket AF_UNIX ; l'échange de données se fera via un fichier local), soit sur des machines différentes reliées par un réseau LAN/WAN (Ex : socket AF_INET pour les réseaux de type TCP/IP).

[1] On appelle « processus » un logiciel en cours de fonctionnement.

Le socket est donc associé à un **domaine** d'utilisation, dont voici quelques exemples :

- AF_UNIX Communication par le système de fichiers Unix
- AF_INET Domaine du protocole IPv4
- AF_INET6 Domaine du protocole IPv6
- AF_AX25 Communication par ondes radioamateurs
- AF_X25 Communication par paquets

4 Créer un socket : Primitive SOCKET

Cette primitive permet d'obtenir un descripteur de socket valide.

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

descr_sock = socket(int Domaine, int type, int protocole);
```

Domaine: C'est le domaine évoqué au chapitre précédent

Type : Il s'agit du type de service associé au socket.

- SOCK_DGRAM est orienté transmission par datagramme. Une transmission par datagramme ne maintient pas une connexion entre le serveur et ses clients. Le message est envoyé mais peut être perdu car aucun acquittement n'est transmis. Ce mode de communication est simple à mette en œuvre et utilise peu de bande passante sur le réseau. Dans le domaine AF_INET, il correspond au protocole UDP.
- SOCK_STREAM correspond à un mode connecté entre un serveur et un client précis. La communication est maintenue jusqu'à ce que le client ou le serveur envoie le signal de fin de connexion. Dans l'intervalle, l'intégrité des données est assurée par un système d'acquittement et de numérotation des messages. Ce mode fiable correspond dans le domaine AF_INET au mode TCP.
- SOCK_RAW correspond au mode « maintenance » et permet de lire ou de générer tout type de trame. Les analyseurs réseau utilisent ce mode.

Protocole : Cet argument est généralement à 0 car il fait double emploi avec les arguments Domaine et type.

4.1 Informations de connexion : les structures de données

En fonction du domaine choisi, il faut maintenant préciser les informations de connexion. Par exemple, pour le domaine AF_UNIX il faut préciser le nom du fichier d'échanges utilisé. Pour le domaine AF_INET, il faudra préciser l'adresse IP du client attendu (INADDR_ANY pour accepter tous les clients), et pour le client, l'adresse IP du serveur.

Ces informations sont regroupées dans des structures de données associées au domaine demandé. Elles sont ensuite associées au socket par les primitives **bind**, **recvfrom, sendto**, ou **connect**, suivant qu'il s'agit du client ou du serveur.

Exemple : Domaine AF_UNIX, structure **sockaddr_un**
Domaine AF_INET, structure **sockaddr_in** et **in_addr**

Note : Le contenu de ces structures est détaillé dans la documentation en ligne, et dans le document « Socket Linux Berkeley » remis précédemment.

4.2 AF_INET : Identifier un lien client – serveur de façon unique

Les adresses IP permettent de distinguer deux machines différentes. Mais sur chaque machine, plusieurs processus peuvent communiquer. Il faut donc identifier de façon unique le processus en vis-à-vis.

Dans le domaine AF_INET (TCP/IP), c'est le **numéro de port** qui remplit cette fonction.

Côté serveur, chaque socket d'écoute est associé à un numéro de port TCP/IP. La liste des numéros de ports normalisés se trouve dans le fichier /etc/services. Le port occupe deux octets dans la trame TCP/IP. Chaque machine peut donc utiliser 65535 ports.

Côté client, un socket est utilisé pour accéder au réseau. Ce socket utilise également un numéro de port, généralement attribué automatiquement au moment de la connexion. Si la machine cliente effectue plusieurs connexions sur le processus serveur (bien sûr dans le cas où le processus serveur accepte de multiples connexions, serveur web par exemple), chaque connexion utilisera un port de sortie différent côté client. Le serveur connait donc le processus client de façon unique : adresse IP et numéro de port.

5 Nommer un socket : primitive BIND

Utilisée côté serveur, cette primitive associe au socket d'écoute les paramètres de la connexion définis dans la structure de données.
Côté client, cette association n'est pas nécessaire car il ne s'agit pas d'un socket d'écoute. Le numéro de port est fixé automatiquement, et la structure de données correspondant au serveur à joindre est utilisée dans la primitive **connect**, et **sendto/recvfrom**.

Exemple, dans le domaine AF_INET, il s'agit de la structure **sockaddr_in** qui contient entre autre l'adresse IP du client attendu (INADDR_ANY pour tous les clients), le numéro du port d'écoute.

```
bind( descr_sock, (struct sockaddr *)&serveur, sizeof(serveur));
```

6 Cas du mode connecté (SOCK_STREAM)

6.1 File d'attente de socket : Primitive LISTEN

Côté serveur, **en mode connecté** (SOCK_STREAM, protocole TCP), il faut définir le nombre de connexions en attente que le processus peut accepter. Au-delà de ce nombre, le client recevra un message lui indiquant que le serveur est occupé. Généralement, 5 est un nombre suffisant.

```
listen (descr_sock, 5);
```

6.2 Attendre une connexion : Primitive ACCEPT

Cette fonction bloque le processus serveur en attente d'une connexion d'un client. Une fois la connexion établie, il reçoit deux informations importantes :

- Un descripteur de connexion qui est le reflet du lien unique IP/PORT CLIENT/SERVEUR
- Une structure de données contenant les informations de connexion du client (IP/PORT source).

```
sock_client = accept (  descr_sock,
                        (struct sockaddr *)&adresse_client
                        &long_client);
```

Le dialogue se fera ensuite en utilisant le descripteur de cette connexion (sock_client), ce qui permet au serveur de se remette en écoute sur son socket (descr_sock). Bien sûr, cela sous-entend utiliser les techniques de programmation appropriées (fork ou thread).

6.3 Se connecter au serveur : Primitive CONNECT

Côté client, cette fonction demande une connexion avec le serveur. En argument, il faut préciser la structure de données contenant les caractéristiques du serveur (adresse IP / Port d'écoute).

```
int  connect(sockfd, (struct sockaddr *)&destination, longueur);
```

6.4 Dialoguer avec le client

En mode connecté, les primitives utilisées sont généralement les classiques READ et WRITE, utilisant le descripteur de socket approprié :

- Côté serveur, le descripteur obtenu par la primitive **connect**,
- Côté client, le descripteur obtenu à la création du socket de sortie.

7 Cas du mode non connecté (SOCK_DGRAM)

Ce mode de fonctionnement non connecté (protocole UDP) demande également un socket valide et nommée, comme décrit aux paragraphes précédents (Socket et Bind).

Les échanges de messages se font sans connexion préalable. La notion de serveur est donc plus floue. Généralement, on considère que le serveur est le premier processus en attente de message (*recvfrom*) car son port d'attente est bien défini. Le client utilise un port libre attribué automatiquement qui sera utilisé par le serveur pour une éventuelle réponse.

Ce système d'échange de messages sans connexion permet généralement d'éviter l'utilisation de programmes multiprocessus ou multithread.

7.1 Attente d'un message : Primitive RECVFROM

Cette fonction contient en paramètre une structure *sockaddr* vide qui recevra les caractéristiques du client qui envoie le message. Cela permet d'envoyer un message de réponse.

```
recvfrom( int socket,              // Descripteur socket d'écoute
          void *buffer,            // Adresse de stockage du message
          size_t length,           //    et taille
          int flags,               // Options
          struct sockaddr *address,      // Coordonnées du client
          socklen_t * address_len );
```

7.2 Envoi de message : Primitive SENDTO

Cette fonction utilise le socket défini auparavant pour envoyer un message à un client dont les coordonnées figurent dans une structure *sockaddr* fournie en argument.

```
sendto(   int sockfd,              // Descripteur de socket
          const void *buf,         // Adresse du message à envoyer
          size_t len,              //     sa taille
          int flags,               // Options
          const struct sockaddr *dest_addr, // Coordonnées du destinataire
          socklen_t addrlen);
```

8 Terminer une connexion : Primitive CLOSE

L'appel de cette fonction supprime le point de connexion. L'argument fourni est le descripteur de socket que l'on souhaite fermer. Cette fonction est utilisée côté client et côté serveur.

9 Fonctions associées

9.1 Conversion de format

La communication par trame fait apparaitre le problème de la représentation des nombres codés sur plusieurs octets : dans quel ordre ces octets seront-ils transmis ? Généralement, la représentation standard retenue est de type *big-endian*. Afin de permettre aux ordinateurs reliés au réseau de s'entendre sur la valeur des entiers multi-octets, il faut définir un arrangement réseau. La bibliothèque netinet/in.h fournit les fonctions suivantes :

```
u_long_int htonl( u_long_int hostlong ) ; // Host to Network Long
u_short_int htons( u_short_int hostshort); // Host to Network short
```

```
u_long_int ntohl( u_long_int netlong); // Network to Host  Long
u_short_int ntohs(u_short_int netshort); // Network to Host Short
```

Par exemple, htons() et ntohs() sont à utiliser pour convertir les numéros de port.

9.2 Conversion d'adresse IP

Plusieurs fonctions permettent de convertir les adresses IP du format W.X.Y.Z au format binaire utilisé sur le réseau (et vice versa). Consulter les manuels en ligne.

```
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
```

Par exemple, *inet_aton()* et *inet_addr()* sont utilisées pour remplir la zone adresse IP de la structure *in_addr* (elle-même contenue dans la structure sockaddr_in) à partir d'une adresse W.X.Y.Z.

Voir également :

```
in_addr_t inet_network(const char *cp);
struct in_addr inet_makeaddr(int net, int host);
in_addr_t inet_lnaof(struct in_addr in);
in_addr_t inet_netof(struct in_addr in);
```

9.3 Résolution de nom et d'adresse IP

Un certain nombre de fonctions permettent d'obtenir les noms d'une machine (Nom DNS) à partir de l'adresse IP ou d'obtenir l'adresse IP à partir du nom de domaine.

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

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

struct hostent *
gethostent(void);

getaddrinfo(const char *hostname, const char *servname,
    const struct addrinfo *hints, struct addrinfo **res);

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

9.4 Résolution de numéro de port et de service

Ci-dessous quelques fonctions permettant d'obtenir le nom du service (/etc/services) à partir du numéro de port et du protocole (udp/tcp) :

```
struct servent *getservent(void);
struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
```

10 Commandes shell

La commande la plus utile pour vérifier l'état des connexions sur une machine est NETSTAT.

Ex : netstat –A inet –alpn Pour afficher toutes les connexions TCP/IP établie ou en attente avec le PID correspondant.

Il est également possible d'espionner le trafic réseau avec des outils comme TCPDUMP ou WIRESHARK.

11 Liens utiles

Winsock pour windows : http://melem.developpez.com/tutoriels/api-windows/winsock/