Chapitre 7

Communication Interprocessus par Socket en C

Dans ce chapitre nous allons restreindre notre étude à la communication Interprocessus en locale et en mode non connecté

BENATTOU MOHAMMED

1. Un point d'accès à la communication

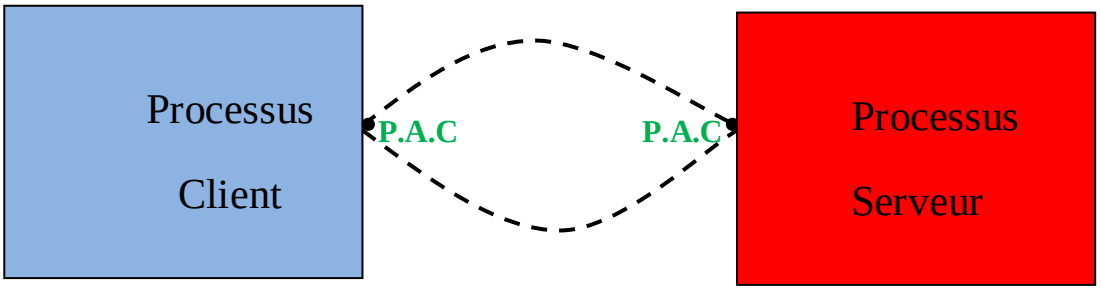

Figure 1 : P.AC Un point d'accès à la communication

- Point d'accès à la communication permet à un processus d'envoyer et de recevoir des informations
- Point d'accès à la communication permet à une machine client ou serveur de communiquer avec l'extérieur.
- En communication distante (Les deux processus se trouvent dans deux machines distante), un point d'accès à la communication est identifié par un numéro de port et une adresse IP. Une communication client/serveur se fait à travers les points d'accès client et serveur.
- Un point d'accès à la communication est défini au niveau implémentation par une API (Application Programming Interface) socket.

La notion de sockets a été introduite dans les distributions de Berkeley d'UNIX, c'est la raison pour laquelle on parle parfois de sockets BSD (Berkeley Software Distribution).

Les sockets se situent juste au-dessus de la couche transport du modèle OSI. Elles utilisent les protocoles de la couche de transport TCP/UDP définissant le mode de communication (connecté, non connecté) et les services de la couche réseau (protocole IP / ARP RARP) définissant les adresses de transports.

Une socket est un point d'accès à la communication par lequel un processus peut émettre ou recevoir des informations à un autre processus se trouvant sur la même machine ou sur une machine distante.

2. CREATION D'UNE SOCKET

La fonction socket permet de créer une Point d'Accès à la communication. Elle prend trois paramètres en entrée

int socket(domaine, type, protocole)

Le domaine : le paramètre domaine spécifie un domaine de communications dans lequel la connexion aura lieu et par la suite le format des adresses qui pourront être données à la socket et les différents protocoles supportés pour la communication. Il existe deux principaux domaines de sockets :

AF_UNIX : le domaine Unix, est utilisé pour une communication Locale, le client et le serveur sont dans ce cas deux processus de la même machine et qui communiquent localement

AF_INET : domaine pour la communication distante, le client et le serveur doivent se trouver dans deux machines différentes.

Le type : décrit la sémantique des communications

- **SOCK_STREAM** : Communication en mode connecté : Support de dialogue garantissant l'intégrité, fournissant un flux de données binaires, et intégrant un mécanisme pour les transmissions de données hors-bande.
- **SOCK_DGRAM** : communication en mode non connecté. Transmissions sans connexion, non garantie, de datagrammes de longueur maximale fixe.
- **SOCK_SEQPACKET** : Dialogue garantissant l'intégrité, pour le transport de datagrammes de longueur fixe. Le lecteur doit lire le paquet de données complet à chaque appel système récupérant l'entrée.
- **SOCK_RAW**
  communication de bas niveau, réservé aux développeurs de nouveaux protocoles

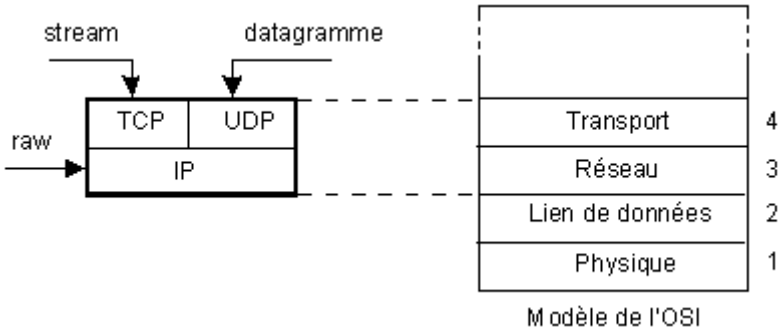

Figure 2 : type d'une socket

Cette figure décrit 3 types de socket suivant le mode d'utilisation mode connecté TCP (**SOCK_STREAM**) et orienté non connecté UDP (**SOCK_DGRAM**) et celui permettant l'accès aux données réseaux (**SOCK_RAW**)

Protocole : permet de spécifier le protocole de communication fournissant le service désiré
Par défaut ce paramètre prendra la valeur 0, suivant le type il affectera le protocole correspondant

La fonction socket retourne un descripteur de socket, de même nature que ceux qui identifient les fichiers dans le système Unix. Il est accessible par le processus créateur et ses Fils. En cas d'erreur la fonction socket() retourne -1.

Exemple : création d'une socket pour une communication distante en mode connecté

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

main(int argc,char *argv[])
{
int idsock;


idsock=socket(AF_INET,SOCK_STREAM,0);

if(idsock==-1)
{
        perror("socket");
        exit(0);
}

}
```

3. Attachement d'une socket à une adresse

Une socket après sa création, n'est accessible que par le processus créateur et ses fils. Un processus distant ne peut accéder à la valeur du descripteur si un mécanisme de nommage de socket n'est pas utilisé. Ce mécanisme est fourni par la fonction *bind* () permettant d'associer à un point de communication une adresse de transport. Chaque processus disant doit connaitre alors l'adresse associé à un point de communication. Le système se charge de retrouver le descripteur à partir de l'adresse de transport.

int bind(int sock, (struct sockaddr *)adresse, int lenght)

La fonction *bind()* permet d'attacher une soket à une adresse de transport

- int sock : descripteur de la socket retourné par la fonction socket()
- struct sockaddr adresse : pointeur sur l'adresse de transport
- int lenght : taille de l'adresse

Le format de l'adresse de transport varie suivant le domaine de communication. La structure *struct sockaddr* est une structure générique à partir de la quelle elle est redéfinie des structures pour une communication locale, distante, privé, bas niveau …

Format des adresses en communication Locale

La famille de socket **AF_UNIX** sert à faire communiquer efficacement des processus sur la même machine. Traditionnellement, les sockets de domaine UNIX peuvent ne pas être nommées ou bien être liées à un chemin d'accès, lequel sera marqué comme étant de type socket, sur un système de fichiers. Une adresse de socket de domaine UNIX est représentée dans la structure suivante :

```
#define UNIX_PATH_MAX    108
struct sockaddr_un {
    sa_family_t sun_family;               /* AF_UNIX */
    char        sun_path[UNIX_PATH_MAX];  /* chemin accès */
};
```

Une socket de domaine UNIX peut être liée, avec bind(), à un fichier dont le chemin d'accès est une chaîne de caractère.
L'exemple suivant permet de créer une socket pour une communication locale avec le mode non connecté. Après création la socket est attachée à un fichier. La commande netstat permet connaitre l'état du réseau, les sockets crées, leurs états ….

```
#include<stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include<string.h>
#include<sys/un.h>
#include<stdlib.h>

main(int argc,char* argv[])
{
        int idsock, idbind;
        struct sockaddr_un adresse;

        idsock= socket(AF_UNIX,SOCK_DGRAM,0);
        if(idsock==-1){
                perror("socket");
                exit(0);
                  }
/* PREPARATION DE L'ADRESSE D'ATTACHEMENT */
         adresse.sun_family=AF_UNIX;
         strcpy(adresse.sun_path,argv[1]);
  idbind= bind(idsock,(struct  sockaddr *) &adresse,sizeof(adresse));
  if(idbind==-1) {
        perror("bind"),
        exit(0); }

}
```

4. Communication en mode UDP

La communication entre processus est en mode non connecté. Le protocole UDP permet une transmission sans connexion. Les données sont envoyées sous formes de paquet indépendamment de toute connexion. C'est un protocole rapide, simple à mettre en œuvre mais moins fiable que le TCP. En effet UDP ne permet de garantir la bonne livraison des datagrammes à destination, ni leur ordre d'arrivée.

Pour réaliser une application client serveur en mode non connecté :

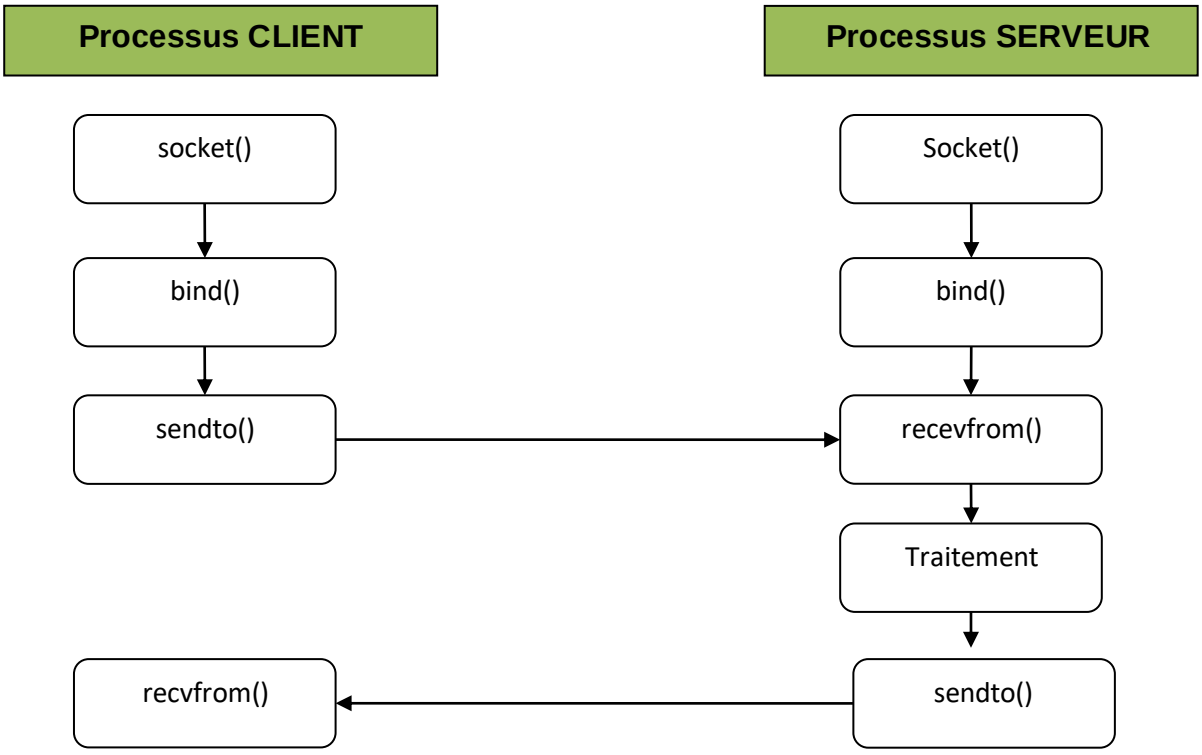

Figure 3 : Communication Interprocessus en mode UDP

Principe de communication

-La partie processus serveur crée une socket et l'attache à une adresse pour qu'elle puisse être accessible de l'extérieur par son adresse. Le processus serveur se met en attente de requête du processus client en appelant la fonction *recevfrom()* qui est une primitive bloquante.

-La partie processus client crée une socket et l'attache à une adresse pour qu'elle puisse être accessible de l'extérieur par son adresse. Elle envoi sa requête par la primitive *sendto()*.

-Une foi que le serveur reçoit la requête, il la traite et envoi le résultat par la primitive sendto().

-Le processus client attend le résultat par appel à la primitive *recevfrom()* .

Pour implémenter une application client serveur en mode UDP deux primitives nous reste à décrire leurs syntaxes et leur fonctionnement

Sendto() : permet d'envoyer un message par une socket (un P.A.C) à un destinataire dont l'adresse est spécifié. En effet on y en mode non connecté le client envoi les données sans demande préalable de connexion au serveur. Le seul moyen permettant d'identifier le serveur et la spécification de son adresse lors de l'envoi.

Syntaxe :

```
int sendto(
  int socket,          /* descripteur de la socket d'emission */
  char* msg,           /* adresse du message a envoyer       */
  int lg,              /* longueur du message                */
  int option,          /* option d'envoi 0 par défaut */
  struct sockaddr* dest;  /* l'adresse de la socket destination  */
  int lgdest           /* longueur de cette adresse          */
);
```

recvfrom() : permet de recevoir un message par une socket (un P.A.C) d'un émetteur dont l'adresse est spécifié. En effet on y en mode non connecté le récepteur ne peut identifier l'émetteur que par son adresse spécifiée à la réception.

```
int recvfrom(
  int socket,          /* descripteur de la socket de réception */
  char* msg,           /* adresse reçu  du message    */
  int lg,              /* taille du  message      */
  int option,          /* option de réception 0 par défaut   */
  struct sockaddr* Exp;   /* l'adresse de l'expéditeur  */
  int* lgdExp          /* taille de l'espace réservé  à l'adresse de l'expéditeur   */
);
```