IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Programmation socket Unix

UE OSAP

Printemps 2020

1 Introduction

Vous connaissez le principe du FIFO. Vous avez noté que l'on ne peut s'envoyer des données que dans un seul sens. C'est d'ailleurs très exactement ce que propose le service des *pipes*. Ne serait-ce pas fabuleux si deux processus Unix s'exécutant sur un même machine pouvaient s'envoyer des données dans les deux sens, exactement comme on le fait avec des sockets réseau ?

Ne perdez plus espoir car voici la réponse : les *sockets du domaine Unix* !

Pour manipuler des sockets réseau vous avez travaillé dans le *domaine* (ou *protocol family*) `PF_INET`. Eh bien il suffit de travailler dans le domaine `PF_UNIX` (voir `man 2 socket`, ainsi que `man 7 unix`). [1]

Globalement, reprenez vos clients-serveurs, et remplacez `PF_INET` par `PF_UNIX`. Bon, cela ne suffit pas tout à fait, car il faut adapter le nommage de notre socket. Désormais on utilise une structure `struct sockaddr_un` (UNIX) à la place de `struct sockaddr_in` (INET) :

```
struct sockaddr_un {
        unsigned short sun_family;  /* AF_UNIX */
        char sun_path[108];
    };
```

Le champ `sun_path` (*socket Unix path*) est un nom de fichier local (un fichier spécial de type socket). Autrement dit, l'*adresse* du serveur est un nom de fichier spécial. Les clients se connectent à ce fichier spécial. (Les sockets Unix peuvent également être *anonymes*, sans nom. C'est le cas typiquement de la socket du client et de celle créée par le `accept`.)

Tapez la commande `netstat -x`. Voyez la liste de sockets Unix en cours d'utilisation sur votre PC. Voyez également le *nom* de ces sockets : c'est un nom de fichier. (Soyez curieux, faites un `ls -l` sur ces noms.) Certaines sockets n'ont pas de nom, ce sont des sockets anonymes.

Attention : ces noms de socket unix sont juste des points de rendez-vous, pour que des clients sachent comment contacter un serveur sur sa socket unix. Ne surtout pas croire que c'est un genre de fichier partagé, avec des données qui sont stockés dedans.

Note : Il existe bien d'autres façons d'échanger des informations entre processus sur une machine. [2] Nous connaissons déjà les *pipes*, avec éventuellement l'extension des *pipes nommés*. Il y a également les services IPC du Système V (*Inter-Process Communication*) : les files de messages, les sémaphores et la mémoire partagée, ou encore leurs variantes Posix. Mais ceci est une autre histoire...

1. Évitez la confusion fréquente entre `PF_INET` et `AF_INET` ! En effet, `PF_INET` (et autres `PF_*`) désigne une famille de protocoles, et s'utilise avec l'appel système `socket()`. En revanche, `AF_INET` (et autres `AF_*`) désigne une famille d'adresses et s'utilise pour construire une adresse de socket (`struct sockaddr`) à passer aux appels systèmes `bind()`, `connect()`, `sendto()`, `recvfrom()`,... Même si ces macros sont définies avec les mêmes valeurs, on ne parle pas tout à fait de la même chose...
2. Et même pour un besoin plus ponctuel de socket Unix on peut se contenter de `socketpair(2)`

2 Exercice proposé

Le but de l'exercice est d'implémenter un service de bus logiciel entre processus.

D'un point de vu pratique, les processus se connectent à ce bus. Ils peuvent alors annoncer des «événements», si bien que lorsque quelqu'un annonce un événement sur le bus, chacun reçoit une copie de cet événement.

Le bus sera en fait implémenté par un processus spécifique (appelons-le un *répéteur*) qui offre une socket Unix en mode serveur (ou *listen*). Les clients, des autres processus, se connectent à cette socket Unix. Lorsqu'un client veut annoncer un événement, il l'envoie sous forme de message au serveur sur cette socket. Le serveur se charge de retransmettre ce message à tous les autres clients connectés (c.-à-d. tout le monde sauf à l'émetteur).

Nous allons procéder étape par étape, ne cherchez pas à résoudre tous les problèmes à la fois.

2.1 Première étape : choix du type de socket

Puisque l'on parle de s'échanger des messages, la socket sera tout naturellement (si si !) orientée *datagramme*. De plus, on a une notion de *connexion*. (En fait, on a bien envie d'utiliser les appels systèmes `listen()`, `accept()`, `connect()`, qui sont très pratiques.)

Fort de ces deux exigences on s'oriente donc vers le protocol `SOCK_SEQPACKET` (qui garantit également l'ordonnancement des messages, même si on n'en demandait pas tant).

Il n'y a à peine quelques années, le man `socket(2)` nous aurait annoncé froidement que ce protocole n'était implémenté pour aucune famille de protocole... (Ce que l'on peut encore lire sous Solaris.) Ce n'est plus tout à fait le cas... Dans la famille `PF_INET` il y a deux candidats potentiels : SCTP (RFC 3286, RFC 4960) et DCCP (RFC 4340). Dans la famille de protocoles `PF_UNIX`, le man `unix(7)` nous apprend que ce protocole est disponible depuis le noyau 2.6.4.

Faisons preuve d'avangardisme, utilisons donc `PF_UNIX/SOCK_SEQPACKET`.

Notons que même si ce type de protocole est encore peu répandu, il y a un réel besoin de longue date pour des communications de type *message* et *orienté connexion*. On utilise alors typiquement deux familles d'astuces pour palier au manque. L'une consiste à utiliser un protocole *orienté connexion* mais de type *flux d'octets*, que l'on structure en *messages* au niveau applicatif avec un format de type `[longueur,message,longueur,message,...]`. L'autre consiste à utiliser un protocole de type *message* mais *sans connexion*, et à spécialiser au niveau applicatif certains messages pour gérer un établissement et une rupture de connexion. Mais c'est tout de même mieux si c'est la couche protocolaire du système d'exploitation qui offre ce type de service.

Réalisation. Reprenez vos programmes de client et serveur TCP et adaptez-les pour utiliser `PF_UNIX/SOCK_SEQPACKET`.

Notez que l'on va se passer des étapes avec `getaddrinfo()`. En effet, cette fonction a pour rôle de faire des traductions d'adresses IP (typiquement via DNS), et est plutôt dédié au domaine réseau. Certaines implémentations fonctionnent dans le domaine `AF_UNIX`, mais on n'en n'a pas vraiment l'utilité. On peut remplir directement les structures d'adresse `sockaddr_un` ; il s'agit juste de recopier la chaîne de caractère avec le nom du fichier socket que l'on s'est choisit.

Vous êtes libre d'utiliser des `read(2)`/`write(2)` ou des `recv(2)`/`send(2)`, voir des `recvfrom(2)`/`sendto(2)`. On vous demande cependant de faire preuve d'une certaine cohérence...

2.2 Seconde étape : écouter tout le monde à la fois

Retournez dans tous les sens votre serveur «multi-processus» et constatez qu'il ne peut pas réaliser le travail demandé. Le serveur multi-processus sait bien gérer plusieurs sockets simultanément (un processus pour la socket qui attend les connections, et un processus par connexion client). Mais dans ce cas les traitements de chaque socket sont très indépendants les uns des autres. Dans l'exercice proposé, ce n'est pas le cas puisqu'un message sur une socket quelconque doit être réémis sur toutes les autres sockets. Pour réaliser cela, il faut non pas plusieurs processus, mais un seul qui soit capable d'écouter toutes les sockets à la fois, et dès qu'il y a un message sur l'une, le renvoyer aux autres.

Pour se mettre en attente sur plusieurs flux d'entrées-sorties (*file descriptors*) à la fois il nous faut un outil spécifique. L'appel système capable de faire ça c'est `select(2)` (ou les variantes `poll(2)` et `epoll(2)` ou encore les variantes `pselect(2)` et `ppoll(2)` qui réagissent également aux signaux).

Pour vous convaincre (si ce n'est déjà fait) de l'intérêt de `select()`, considérez simplement votre client. Lui non plus ne peut répondre au cahier des charges. Son comportement actuel est le suivant : il attend que l'utilisateur tape quelque chose sur `stdin` pour l'envoyer sur la socket, puis il attend quelque chose sur la socket qu'il affiche, puis reboucle. Dans l'exercice proposé, un client peut recevoir plusieurs messages sur sa socket avant d'envoyer quelque chose, et réciproquement. Il faut que lui aussi soit capable d'écouter à la fois sur `stdin` et sa socket... d'où un `select()`.

Réalisation. Récupérez les fichiers `bus_server.c` et `bus_client.c` dans l'archive `ProgSockUnix.tgz` sur Moodle. Ce sont des listings «à trous». Remplissez les trous. Aidez-vous du man ![3] Lisez également la suite du sujet.

2.3 Consignes

Ces programmes (`bus_server` et `bus_client`) requirent un argument en ligne de commande qui est le nom de la socket Unix. Le serveur requière un second argument qui est le nombre maximum de clients.

Le serveur affiche sur sa sortie standard les messages qu'il reçoit. Il indique à quels clients il les renvoie.

Les clients affichent sur leur sortie standard les messages qu'ils reçoivent. Les messages envoyés sont constitués du texte tapé au clavier, sur l'entrée standard.

Hypothèses simplificatrices. Dans un but hautement pédagogique nous allons faire quelques hypothèses. On considère que l'on est capable de lire un message en une seule fois : non seulement on utilisera un buffer assez grand (choisissez la taille que vous voulez mais tapez des messages qui rentrent), mais on suppose aussi que les opérations de lecture ne seront pas interrompues par des signaux ou autres événements embarrassants. On considère également que l'on est toujours capable de faire les écritures (et en un seul coup également).

Ces hypothèses nous simplifient la programmation, mais ne sont pas forcément vérifiées dans la vraie vie. Notre bus n'est donc pas très robuste. [4] Imaginez des situations où cela peut provoquer des dysfonctionnements.

Expliquez (sans le programmer) ce qu'il faudrait faire si l'on ne posait pas ces hypothèses.

Interopérabilité : testez le bus de votre voisin (clients et serveurs).

3. Consultez le man de `select_tut(2)`.
4. De plus, est-ce que l'on respecte bien les règles de bonne programmation du `select_tut(2)` ?

Et pour aller plus loin : le mécanisme de sockets permet de s'échanger des *données de contrôle* (ou *ancillary data*), en plus de données applicatives. Pour des sockets réseau, cela peut être utilisé pour lire ou positionner des options de socket valables pour un message seulement et non pas pour toute la durée de vie de la socket. Pour des sockets Unix c'est utilisé pour connaître les permissions (*user/group id*) du processus avec qui l'on discute (`man credentials`). On peut également se passer des descripteurs de fichiers d'un processus à l'autre (par exemple, sur notre bus, un processus peut être un *helper* qui ouvre pour les autres certains fichiers auxquels les autres processus n'ont pas normalement accès). Si le sujet vous intéresse, lisez les man de `sendmsg` `recvmsg` et `cmsg`.