Université Côte d'Azur
Faculté des Sciences - Département d'Informatique

Licence d'Informatique L2
Introduction aux Systèmes et Réseaux

TP n ° 5 : Client-serveur, `select`, réseaux

L'objectif de ce TP est la conception d'une architecture client-serveur efficace, pouvant par exemple être utilisée pour la création finale d'un serveur de *chat* dans le TP suivant.

1 Serveur synchrone mono-client, avec auto-connexion

En TD, nous avons vu une architecture client-serveur basée sur les *tubes*, à l'aide de la primitive `mkfifo`. Dans l'exemple du TD, le serveur (mono-client), créait les deux tubes de communication client vers serveur et serveur vers client, et avait la charge de les supprimer toutes les deux, une fois le travail terminé. Dans une application classique (multi-client), le serveur ne connaît pas au départ le nombre de clients qui se connecteront à lui. C'est donc aux clients, et non au serveur, de créer leur propre FIFO de communication serveur vers client. Le serveur ne gérant que le canal de communication clients vers serveur.

Implémenter un `echo_server.py` dont le travail est de renvoyer au client le texte reçu. `echo_client.py` doit envoyer au serveur les données lues sur l'entrée standard, et écrire sur la sortie standard les données reçues du serveur.

Le serveur doit :

1. Créer sa propre FIFO (donner un nom unique dans `/tmp/`, en intégrant le PID du serveur dans le nom,
2. Programmer une suppression de celle-ci à la terminaison du processus (module `atexit`),
3. Ouvrir sa FIFO en lecture,
4. Lire les données envoyées par le client (ces données doivent être le chemin de la FIFO du client),
5. Ouvir la FIFO client en écriture,
6. Faire le travail prévu (appel de la fonction `server` du TD).

Le client doit :

1. Créer sa propre FIFO (donner un nom unique dans `/tmp/`, en intégrant le PID du serveur dans le nom,
2. Programmer une suppression de celle-ci à la terminaison du processus (module `atexit`),
3. Récupérer le chemin de la FIFO serveur comme 1er argument de la ligne de commande et l'ouvrir en écriture,
4. Envoyer le chemin de sa propre FIFO au serveur,

5. Ouvrir sa propre FIFO en lecture,
6. Faire le travail prévu (appel de la fonction `client` du TD).

Note : Attention à l'ordre des ouvertures de FIFO en lecture et écriture. L'ouverture en lecture d'un côté est *bloquante* tant que l'autre côté n'a pas ouvert en écriture. I.e. vérifier que le client et le serveur ouvrent en premier l'une des deux FIFO, puis l'autre. Sinon, risque d'interblocage.

Note 2 : Une lecture depuis un tube fermé à l'autre bout (car le processus écrivain s'est par exemple terminé) retournera un séquence d'octets de taille nulle, i.e. une fin de fichier. Une écriture dans un tube fermé à l'autre bout provoquera l'envoi par le noyau d'un signal `SIGPIPE` qui provoquera la levée d'une exception `broken pipe`. Il faut donc veiller à fermer les deux extrémités d'un tube dans le bon ordre.

2 Serveur synchrone multi-clients

Maintenant que l'on a vu comment un client peut s'auto-connecter au serveur, implémenter une version multi-client. Un seul serveur gère les echos de plusieurs clients. Les clients en revanche ne se connaissent pas les uns les autres. Le serveur communique vers un client donné au travers de la FIFO de ce client. En revanche, tous les clients communiquent vers le serveur au travers de l'unique FIFO serveur. Il faudra donc identifier les messages. Pour cela :

— Les clients doivent préxixer leurs messages par leur identifiant.
— Le serveur va recevoir de façon impromptue, deux types de messages : des demandes de connexion contenant le chemin vers la FIFO client (auquel cas il doit enregistrer ce nouveau client et se préparer à communiquer avec lui), et des messages "normaux" à renvoyer. Il faut donc que le serveur sache reconnaître ces deux types de messages. Il doit en outre conserver une collection de clients actifs. Un dictionnaire *identifiant client* $\Rightarrow$ *descripteur de fichier client* est suggéré. À chaque connexion d'un nouveau client, une entrée est ajoutée au dictionnaire.

Dans un second temps, faites en sorte qu'à chaque terminaison d'un client (frappe de `^D`), le client soit supprimé du dictionnaire serveur.

Dans un troisième temps, faites en sorte qu'à chaque interruption d'un client (frappe de `^C`), le client soit également supprimé du dictionnaire serveur.

3 Serveur asynchrone multi-clients

Maintenant, nous souhaiterions qu'un client A puisse communiquer avec un client B. En mode synchrone, avec des `read` bloquants, nous sommes confrontés à un problème de taille : si un client A veut communiquer avec un client B, lors de l'émission du message par A, il n'y aura aucun problème. En revanche, c'est lors de la réception du message par B qu'il y aura un problème insoluble : B est normalement bloqué en lecture sur son entrée standard (car il attend des donnée depuis le clavier), et non sur sa FIFO : les messages provenant de A ne seront donc lus qu'une fois que des données auront été lues sur l'entrée standard de B (puis envoyées au serveur), et que B se mettra à nouveau à l'écoute de sa FIFO.

Pour résoudre ce problème, il faudrait que les clients soient en même temps à l'écoute de leur entrée standard et de leur FIFO. C'est là qu'entre en jeu la primitive `select()`, localisée en python dans le module `select`.

```
active_readers, active_writers, _ = select.select(readers_list, writers_list, _)
```

`select` prend trois arguments. Dans notre cas, seul le premier présente un intérêt. Dans quelques autres cas, le second peut aussi être utile. Ces arguments sont des listes de *descripteurs de fichiers* (ou d'*objets de type fichier*) ouverts respectivement en lecture et en écriture que l'on désire "surveiller". `select` est une fonction bloquante qui va retourner une liste des descripteurs de fichier qui sont "prêts". Elle permet donc de surveiller plusieurs descripteurs de fichier, attendant qu'au moins l'un de ceux-ci devienne "prêt" pour certaines classes d'opérations d'entrées-sorties. Un descripteur de fichier est considéré comme prêt s'il est possible d'effectuer l'opération d'entrées-sorties correspondante (par exemple, un read() si c'est un descripteur d'un fichier ouvert en lecture) sans bloquer.

Nous allons modifier donc l'application serveur a minima de façon à ce que tout message envoyé depuis un client soit transmis par le serveur à tous les clients (message multicast *@everyone* d'un *chat*).

En ce qui concerne le client, il faudra ré-architecturer la fonction `client` en y apportant les modifications suivantes :

— Toujours dans une boucle infinie, effectuer un `select` sur les 2 descripteurs de fichiers ouverts en lecture qui nous intéressent (0 et la FIFO client),

— Parcourir la liste des descripteurs prêts (boucle `for`) et effectuer les actions correspondantes.