M2 FEADéP – Systèmes et réseaux
TP5 Sockets II

Nicolas Chappe
École Normale Supérieure de Lyon

1 Système de fichiers

Avant de retourner à notre programme de discussion, parlons un petit peu de système de fichiers.

Exercice 1

La commande `stat` donne des informations précises sur un fichier. Lancez-la sur divers fichiers de votre dossier personnel et de votre système et essayez de comprendre la signification de chacun des champs. Quelques fichiers système intéressants : / /dev/ /bin/sudo /proc/self

Note. Les pages de man de `stat` et `inode` pourront vous éclairer.

2 Complément sur les sockets

Voir le sujet de TP précédent pour des indications sur comment manipuler les sockets en C.

On peut utiliser `read` et `write` sur les sockets comme sur tout descripteur de fichier, mais des fonctions spécifiques aux socket offrent plus de flexibilité : `recv` et `send`. Ces deux fonctions prennent un paramètre supplémentaire `flags` qui peut notamment prendre la valeur `MSG_DONTWAIT` pour que les entrées/sorties ne soient pas bloquantes.

3 Messagerie instantanée en réseau

On souhaite étendre notre logiciel de messagerie instantanée du TP3 pour que des clients puissent s'y connecter en réseau. Le code Python d'un client est fourni dans le fichier `client.py`, il n'y aura pas besoin de le modifier.

Exercice 2

La communication entre client et serveur doit s'effectuer selon un *protocole* commun. Par souci de simplicité, ce protocole consistera simplement à envoyer des objets `Message`. Mais pour cela il faut que le client et le serveur soient d'accord sur la taille d'un message, car time_t peut faire 4 ou 8 octets selon le système [1]. Avant de commencer à réellement implanter la communication client/serveur, nous devons donc remplacer le time_t par un uint32_t de `stdint.h` stocké en *big endian* (utiliser `htonl/ntohl` pour la conversion).

Exercice 3

Quand l'option `--serveur` est passée à notre programme de discussion on lancera un thread supplémentaire qui gèrera la communication avec des clients via des sockets TCP.

Lorsqu'un client se connecte (`accept`), ce thread créera un nouveau thread chargé d'envoyer au client les nouveaux messages au fur et à mesure, le code sera quasi-identique à celui de la fonction du TP3 qui affiche les messages sur la sortie standard, sauf qu'au lieu d'écrire le message formaté avec `printf`, le ou les `Message` bruts devront être écrit dans le fd pour le communiquer au client.

Le fichier `serveurtcp.c` fourni constitue un fragment de code minimal pour un thread faisant office de serveur TCP, vous pouvez copier-coller son contenu pour la suite de ce TP.

Exercice 4

Testez la communication entre votre serveur et le client Python. Lancez tout d'abord le serveur, puis lancez un client et donnez-lui comme adresse du serveur `localhost` pour vous connecter à votre propre machine. Vous pouvez aussi tester avec plusieurs clients.

Pour tester en réseau (ce n'est pas indispensable), vous pouvez obtenir l'adresse IPv4 d'une machine à l'aide de la commande `ip addr`.

Exercice 5

Une fois la communication du serveur vers les clients soigneusement testée, il reste à implanter la réception des messages provenant des clients. Au moment du `accept`, créez un second thread avec cette responsabilité.

Exercice 6

Assurez-vous que les threads de lecture/écriture liés à une connexion se terminent bien une fois la connexion close par le client.

`read` et `recv`, quand ils sont non-bloquants, renvoient 0 quand la connexion a été close. Quant à `write` et `send`, ils renvoient `-1` et donnent la valeur `EPIPE` à `errno` (voir `man 3 errno`).

1. Chercher "bug de l'an 2038" pour le contexte.

Une difficulté se pose ici, tenter d'écrire dans un socket (ou un pipe) dont la connexion a été close déclenche en plus l'envoi du signal `SIGPIPE` au programme, qui par défaut l'arrête brusquement. Pour contourner le problème, utiliser `send` plutôt que `write` et lui passer le fanion (*flag*) `MSG_NOSIGNAL` [2]. Alternativement, ajouter `signal(SIGPIPE, SIG_IGN)` en début de `main` désactive globalement le signal SIGPIPE.

Exercice 7 (avancé)

Plutôt que de créer des threads à chaque connexion, il serait possible d'avoir un thread s'occupant de tous les envois de données vers les clients, et un thread s'occupant de toutes les réceptions de données venant des clients. Un tableau de descripteur de fichiers pourrait être commun à ces deux threads, par exemple via une variable globale.

Le thread chargé des écritures a simplement à écrire dans chacun des sockets connectés aux clients quand il doit envoyer un message.

Pour le thread chargé de lire les sockets connectés aux clients, l'appel système `poll` permet de surveiller un ensemble de descripteurs de fichiers et d'attendre qu'un évènement se passe pour l'un d'entre eux. Ici, on s'intéressera à l'évènement `POLLIN` qui survient quand il y a des données disponibles à la lecture dans un descripteur de fichier.

2. difficile à trouver, il est dans la page de man de socket en section 7