iUT ORSAY
Département Informatique

DUT Informatique
Module Système S4 C
2009 / 2010

Travaux Pratiques n° 5 : Sockets Stream

Nom(s) :
Groupe :
Date :

Objectifs : manipuler les primitives relatives à la communication par sockets stream, être capable d'implanter des systèmes client-serveur sur ce mode de communication.

1 Avant propos : messagerie instantanée ou *chat*

La technologie de messagerie instantanée, ou *chat*, permet l'échange quasiment en temps réel de messages textuels entre différents ordinateurs d'un même réseau informatique (typiquement l'Internet). Ce type d'application, qui remonte au temps des premiers systèmes multi-utilisateurs (milieu des années 60), est basé sur l'architecture client/serveur. Le fonctionnement général est le suivant : un logiciel client se connecte à un serveur de messagerie instantanée, une fois la connexion établie, lorsque le serveur reçoit un message de la part d'un client, il le relaie à tous ses clients connectés. Les clients peuvent donc communiquer intéractivement entre eux. Tous les clients peuvent lire les messages envoyés par tous les autres. Dans les applications évoluées, on dispose de la liste des clients connectés ainsi que de leur disponibilité pour discuter.

Pourquoi le mode connecté (TCP avec l'utilisation des sockets stream) est adapté à ce type d'application ?

Le but de ce TP est de réaliser une application de chat simple par internet en utilisant les sockets stream. Un exemple de programme avec son serveur et son client vous est proposé sur la page du Système S4 `http://www.lri.fr/~bastoul/systeme`. Le serveur se lance par la commande `./chat_serveur` (sans argument). Un client se lance par la commande `./chat_client` *`IP_serveur pseudo`*, où *`IP_serveur`* est l'adresse IP de l'ordinateur où fonctionne le serveur et *`pseudo`* est une chaîne de caractères correspondant au pseudonyme que souhaite utiliser le client.

Testez l'application. Quel port utilise cette application pour communiquer (utilisez la commande `netstat` -et sa page de man- pour le découvrir) ? Quelle option de `netstat` avez-vous utilisé ?

De combien de processus l'application serveur est-elle composée pour gérer 1, 2, `n` clients (utilisez la commande `ps` -et sa page de man- pour le découvrir) ? De combien de processus est composée une application client ?

2 Implantation de la partie client

Le rôle du programme client est tout d'abord d'établir une connexion avec le serveur de messagerie instantanée. Une fois que la connexion est réalisée, les réceptions et émissions d'informations (purement textuelles) s'effectuent de manière interactive. D'une part, le client attend des informations en provenance de l'utilisateur sur l'entrée standard : l'utilisateur tape une chaîne de caractères sur le terminal et lorqu'il appuie sur la touche « entrée », la chaîne est envoyée au serveur *via* la socket de communication. D'autre part, le serveur peut à tout moment envoyer sur la socket de communication des messages en provenance des autres clients. Les messages doivent s'afficher dès reception.

Proposez une organisation de votre programme (avec un schéma si besoin) pour que les tâches de communications interactives puissent se dérouler en parallèle.

Implantez un programme client pouvant communiquer avec le serveur fourni en exemple en vous aidant des indications ci-après et du squelette *client.c* fourni. **Vous joindrez le code source à votre compte-rendu**.

Indications :
La communication s'effectue de manière très simple : seul du texte est échangé. Le client envoie des chaînes de caractères qui commencent par le pseudo de l'utilisateur. Le serveur envoie les chaînes de caractères qu'il a reçues des différents clients. Les conseils *à suivre* suivants vous faciliteront la tâche :
– Compilez votre programme régulièrement (après chaque étape de l'algorithme) en utilisant l'option `-Wall` de `gcc`, qui vous donnera le maximum d'avertissements.
– Testez systématiquement le code de retour des fonctions appelées pour détecter et comprendre les erreurs à l'exécution. En cas d'erreur, utilisez la primitive `perror()` pour obtenir le maximum d'informations (voir page de man associée ou TP 3).
– Pour lire une chaîne de caractères complète (et non seulement un mot avec `scanf()`), utilisez la primitive `fgets()` (voir page de man associée).

3 Implantation de la partie serveur

Le serveur d'une application dédiée à la messagerie instantanée est un programme relativement complexe. Ce serveur doit gérer autant de connexions qu'il y a de clients (sur des sockets de service) et de plus être en permanence en attente de nouvelles connexions (sur la socket d'écoute). Pour être en mesure de gérer les connexions en parallèle, un processus est dédié à chacune d'entre elles : dès qu'une connexion est acceptée par le serveur, un nouveau processus est crée et c'est ce processus qui assurera la gestion de la communication avec le client. Le processus père se charge quant à lui de l'acceptation de nouvelles connexions entrantes et de la transmission des messages vers l'ensemble des clients (lui seul connaît tous les processus fils ou toutes les sockets de service).
Nous allons construire l'application serveur en plusieurs versions, de la plus simple (qui ne l'est déjà pas tant) à la plus complexe.

3.1 Première version du serveur : délégation partielle

Dans un premier temps, nous allons construire une application simplifiée mais peu élégante : le processus serveur père va déléguer uniquement la gestion de la **réception** sur les sockets de service à des processus fils. Lorsqu'un processus serveur fils recevra une information, il la transmettra au processus serveur père *via* un tube. Le père se chargera lui-même de retransmettre le message à tous les clients (y compris, pour des raisons de simplicité, au client qui a envoyé le message). Le schéma des communications est décrit en Figure 1.

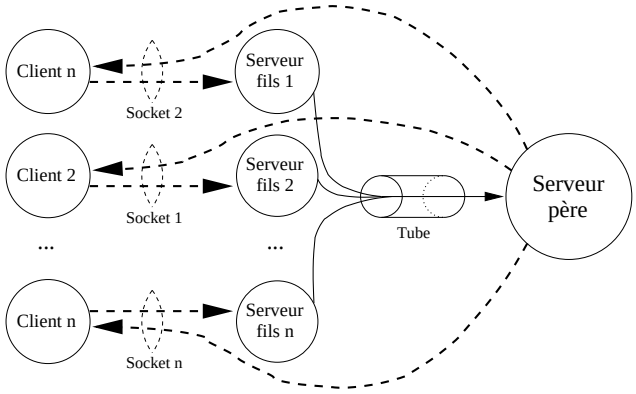

FIGURE 1 – Schéma des communications dans la version à délégation partielle

La principale difficulté à laquelle le processus serveur père doit faire face est de réaliser le double travail d'attente de nouvelles connexions sur la socket d'écoute et de messages à retransmettre sur le tube. Il y a plusieurs manières de traiter cela, en voici deux :

– Le processus serveur père peut recourir à la primitive `select()` qui lui permet d'attendre des informations sur plusieurs descripteurs à la fois. C'est l'objet du TD 4 *multiplexage* qui ne devrait pas avoir encore été traité au début de ce TP. On n'utilisera pas encore cette solution.
– Le processus serveur père peut effectuer normalement son travail d'attente de nouvelles connexions et vérifier seulement périodiquement (en déroutant le signal `SIGALRM`) la présence dans le tube d'informations à retransmettre et en assurer la retransmission si besoin. On a alors besoin de rendre la lecture sur le tube non bloquante, ce qui est possible par un appel à la primitive `fcntl()`, par exemple par l'instruction `fcntl(tube[0],F_SETFL,O_NDELAY);` (voir page de man relative à la primitive `fcntl()`). On utilisera cette solution dans cette version.

D'autres difficultés existent, liées en particulier à la déconnexion des clients. Tout d'abord, lorsqu'un client se déconnecte, le processus serveur fils s'arrête. Il envoie donc un signal `SIGCHLD` à son père (qui l'ignore par défaut) pour lui signifier qu'il meurt. Si le père ne traite pas ce signal, le fils devient un zombie tant que le père n'a pas utilisé la primitive `wait()` (ou `waitpid()`) pour être averti de la mort de son fils.

Pourquoi est-il important d'éliminer les zombies ? Comment procéder pour cette application ?

Une deuxième difficulté liée à la déconnexion est que le processus serveur père risque de chercher à envoyer des messages sur des sockets fermées (à moins qu'un protocole soit mis en place entre le processus serveur père et ses fils lors d'une déconnexion, mais on ne le demande surtout pas). Cela le tuera.

Pourquoi le processus serveur serait-il tué en écrivant sur une socket fermée (regardez la page de man de la primitive `write()` et en particuler l'erreur `EPIPE`) ? Proposez une solution pour que cela n'arrive pas.

Implantez un programme serveur pouvant communiquer avec votre client en vous aidant du squelette *serveur.c* fourni. Ce programme subira au moins une évolution avant d'être rendu.

3.2 Deuxième version du serveur : délégation totale

La première version du serveur était fonctionnelle et relativement simple. Pour autant elle n'était pas vraiment élégante à cause de la délégation uniquement partielle aux processus fils des communications avec les clients. Dans une seconde version plus propre, on souhaite que le processus serveur père se dégage de toute communication directe avec les clients. Lorsqu'un message est envoyé par un client, le processus serveur fils associé envoie ce message au travers d'un tube au processus serveur père (comme dans la première version). Le serveur père (seul à connaître tous les fils) va envoyer ce message aux différents fils au travers d'autres tubes comme montré sur le schéma des communications en Figure 2. Chaque fils retransmettra alors ce message vers le client dont il a la charge.

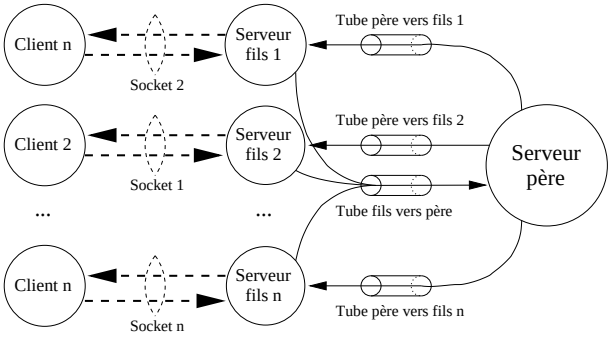

FIGURE 2 – Schéma des communications dans la version à délégation totale

Comment les processus serveur fils vont-ils pouvoir assurer leurs communications interactives (on ne peut prévoir quand le processus serveur père ou le client demanderont de transmettre un message, il faudra donc gérer le tube et la socket en parallèle) ?

Modifiez votre programme serveur pour implanter la délégation totale. Vous prendrez bien garde à ne pas laisser des descripteurs inutilement ouverts. **Vous joindrez le code source à votre compte-rendu**.

3.3 Troisième version du serveur : version ultime

La seconde version est satisfaisante, mais les clients, les serveurs fils et le serveur père doivent tous gérer simultanément deux descripteurs et utilisent pour cela des solutions trop lourdes ou inadaptées. Les clients doivent gérer simultanément l'entrée standard et une socket, les serveurs

fils doivent gérer simultanément un tube et une socket, le serveur père doit gérer simultanément la socket d'écoute et un tube. Clients et serveurs fils utilisent deux processus pour gérer le trafic en parallèle, ce qui est une solution lourde, le serveur père utilise le signal `SIGALRM`, ce qui est assez inadapté.
La solution pour un travail sur si peu de descripteurs est d'utiliser la primitive `select()`, sujet du TD 4 *multiplexage* (ce ne serait par contre pas adapté au traitement de toutes les connexions dans le serveur car elle sont potentiellement nombreuses).

Modifiez votre programme serveur pour que clients et serveurs fils n'utilisent plus qu'un processus et que le serveur père n'ait plus à utiliser le signal `SIGALRM`, grâce à la primitive `select()`. **Vous joindrez le code source à votre compte-rendu**.

Commentaires personnels sur le TP (résultats attendus, difficultés, critiques etc.).