UNIVERSITÉ PARIS SUD
IUT D'ORSAY

**DUT Informatique**
**M3101 Système S3**
**2015 / 2016**

**Travaux Pratiques n° 3 : Sockets stream**

---

Nom(s) :
Groupe :
Date :

---

*Objectifs : comprendre les principes et les mécanismes de communication par sockets stream, être capable de réaliser des systèmes client-serveur sur ce mode de communication.*

**Avant propos.** 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 a accès à plusieurs sessions de chat, publiques ou privées, à des fonctions de modération, etc. (Voir le protocole de *Internet Relay Chat*, RFC 1459 and RFC 2812).

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. Le code exécutable de client et serveur et les squelettes des programmes *client.c* et *serveur.c* vous sont fournis dans l'espace du cours sur Dokeos. 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) ?

**Exercice 1 : Implémentation 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 lorsqu'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 réception.

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

Implémentez 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 :

— 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.
— Pour lire une chaîne de caractères complète (et non seulement un mot avec `scanf()`), utilisez la primitive `fgets()` (voir la page de man associée).

**Notes de cours : Multiplexage.** La manière traditionnelle de construire une application serveur réseau est de construire un bloc principal attendant une connexion par la primitive `accept()`. Dès qu'une demande de connexion survient, on crée un processus fils par `fork()` : le fils prend en charge la communication et le père se remet en position d'attente d'une nouvelle connexion.

La primitive `select()` permet au contraire de construire un serveur autour d'un seul processus qui *multiplexe* les demandes de connexion et qui les prend en charge aussi bien qu'il le peut. L'avantage de l'utilisation de `select()` est que le serveur n'est qu'un seul processus, il n'y a donc pas besoin de primitives de synchronisation ou de communication entre processus. Le principal désavantage est bien sûr de devoir gérer plusieurs connexions en même temps alors que chaque processus travaillait avec une seule connexion dans la solution utilisant `fork()`.

La primitive `select()` est basée sur le concept de `fd_set` qui sont des ensembles de descripteurs de fichiers (*File Descriptor Sets*). On manipule ce type de données en utilisant les macros standards suivantes :

```
fd_set ensemble;
FD_ZERO(&ensemble);              /* Vide ensemble. */
FD_SET(descripteur,&ensemble);   /* Ajoute descripteur a ensemble. */
FD_CLR(descripteur,&ensemble);   /* Retire descripteur de ensemble. */
FD_ISSET(descripteur,&ensemble); /* Vrai si descripteur dans ensemble. */
```

Son prototype est le suivant :

```
#include <sys/time.h>
#include <sys/select.h>
int select(
  int descripteur_max,
  fd_set * ensemble_lecture,
  fd_set * ensemble_ecriture,
  fd_set * ensemble_exceptionnel,
  struct timeval * delai);

struct timeval {
  long tv_sec;  /* secondes */
  long tv_usec; /* microsecondes */};
```

1. `descripteur_max` : le numéro du plus grand descripteur des trois ensembles, plus 1. Ce nombre ne peut pas dépasser la constante `FD_SETSIZE`, dont la valeur varie entre les systèmes et est égale à 1024 sous Linux.
2. `ensemble_lecture` : pointeur vers l'ensemble de descripteurs à examiner en lecture (`NULL` est l'ensemble vide).
3. `ensemble_ecriture` : pointeur vers l'ensemble de descripteurs à examiner en écriture (`NULL` est l'ensemble vide).
4. `ensemble_exceptionnel` : pointeur vers l'ensemble de descripteurs à examiner pour un état exceptionnel, par exemple les messages urgents (`NULL` est l'ensemble vide).
5. `delai` : pointeur vers une structure `timeval` donnant le délai d'attente maximum. Mettre `NULL` pour un temps infini.

Cette primitive renvoie le nombre de descripteurs *prêts* et les trois ensembles sont modifiés pour contenir les ensembles de descripteurs prêts (cela signifie aussi qu'il faudra remettre les ensembles comme il faut avant un nouvel appel). Elle est bloquante jusqu'à ce que le délai d'attente soit fini ou qu'un des événements attendus se produise. Sous Linux, `delai` est aussi modifié pour contenir le temps qui reste jusqu'à la fin du délai initial. En cas de réception d'un signal par le processus, `select()` renvoie $-1$.

**Exercice 2 : Implémentation 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).

La principale difficulté à laquelle le processus serveur 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 les sockets de service. Pour traiter cela, le processus serveur doit recourir à la primitive `select()` qui lui permet d'attendre des informations sur plusieurs descripteurs à la fois.

Expliquez pourquoi l'architecture traditionnelle de serveur qui consiste à générer un processus fils par connexion est difficile à réaliser dans le cas d'une application *chat*.

Une deuxième difficulté, liée à la déconnexion d'un client, est que le processus serveur risque de chercher à envoyer des messages sur des sockets fermées. Cela le tuera.

Pourquoi le processus serveur serait-il tué en écrivant sur une socket fermée ? Proposez une solution pour que cela n'arrive pas.

Implémentez un programme serveur pouvant communiquer avec votre client en vous aidant du squelette *serveur.c* fourni. **Vous joindrez le code source à votre compte-rendu**.