# TP 11

# Sockets, la suite

**Durée prévue :** 1 séance (1h30)

Le but est de terminer et d'améliorer le TP de la dernière fois. **Nous vous fournissons une nouvelle archive, avec des fichiers mis à jour.** L'archive fournie pour le TP contient un répertoire `echoserver2` avec : les sources de la bibliothèque `socklib`, les sources `echoserver.cpp` et `echoclient.cpp` dans lesquelles vous allez travailler, ainsi qu'un `Makefile`.

Un code de serveur vous est fourni : il s'agit de `echoserver.cpp`, à partir duquel le `Makefile` fabrique `echoserver`. Ce serveur réalise le comportement que devait avoir votre serveur du TP précédent.

Si vous aviez fait l'exercice 2 du TP précédent, vous pouvez passer l'exercice 1 du présent TP, soit après avoir adapté votre client, soit en utilisant `telnet` comme client.

Une fonction équivalente à votre fonction `recv_line()` du dernier TP vous est fournie :

```
int recv_line(int sd, std::string &line, char c = '\n');
```

On suppose que, lors de l'appel à cette fonction, `sd` est une socket de dialogue, avec un hôte à l'autre bout qui envoie des caractères ; `c` désigne le caractère de fin de chaîne (qui sera stocké dans la chaîne lue dans `line`). La fonction retourne le nombre de caractères lus en cas de succès, 0 si la socket a été fermée par le client, -1 en cas d'échec.

## EXERCICE 1 ► Un client pour notre serveur

Dans cet exercice, vous devez réaliser l'écriture d'un client pour pouvoir interagir avec le serveur, et prendre la place de `telnet` que nous avons utilisé jusqu'ici. Le code à compléter du client est dans le fichier `echoclient.cpp`, et le `Makefile` le compile en `echoclient`.

**1**) Au niveau du `TODO1` dans le code fourni, modifiez le code afin que le client tente de se connecter à l'adresse qui doit être passée comme premier argument du programme `echoclient`, et sur le port passé comme deuxième argument du programme. En cas de succès, `sd` doit être une socket utilisable pour dialoguer avec le serveur ; en cas d'échec, afficher un message d'erreur et terminer le programme. Vous utiliserez pour cela la fonction `create_client_socket()` au sujet de laquelle `socklib.h` contient des explications.

**2**) Après la connexion au serveur, le client entre dans une boucle *a priori* infinie dans laquelle, à chaque itération, il va attendre une ligne de la part du serveur, puis lire une ligne au clavier, puis l'envoyer au serveur. Le client commence donc par attendre la réception d'une ligne de texte : pourquoi ?

**3**) Au niveau du `TODO2`, modifiez le code de façon à ce que le client attende la réception d'une ligne de texte terminée par `'\n'` de la part du serveur, puis l'affiche sur sa sortie standard.

**4**) Le client lit une ligne entrée par l'utilisateur sur son entrée standard grâce à `getline(cin, line)`. A niveau du `TODO3`, modifiez le programme de façon à ce que, si l'utilisateur entre la commande `quit`, alors le client envoie cette commande au serveur puis sorte que la boucle `while(1)` pour se terminer. Attention, la chaîne obtenue grâce à `getline()` ne se termine pas par un `'\n'`, alors que le serveur attend précisément un `'\n'` pour marquer la fin d'un message.

**5**) Il ne reste plus qu'à modifiez le code au niveau du `TODO4` pour que, s'il n'est pas sorti de la boucle, le client envoie la ligne `line` au serveur. Attention, tout comme le client, le serveur attend toujours une ligne qui se termine par un `'\n'` !

**6**) Compilez votre programme, et testez-le. Comme au TP précédent, lancez le serveur avec `./echoserver 8085`, puis dans un autre terminal lancez votre client avec `./echoclient localhost 8085`. Vous devez obtenir le même comportement qu'avec `telnet`, sinon, il faut déboguer.

## EXERCICE 2 ► Accueil successif des clients

Jusqu'à présent, votre serveur ne traite qu'un seul client, puis se termine : cela ressemblerait tout de même plus à un « vrai » serveur s'il était capable de gérer plusieurs clients... Dans cet exercice, vous allez modifier le code du serveur pour qu'il puisse accueillir plusieurs clients *successivement* : dans cette solution, les clients sont gérés les un après les autres. Pour cela, il suffit qu'après avoir traité un client, le serveur revienne se mettre en attente du client suivant avec la fonction `accept_connection()`. Vous devez donc mettre place une boucle dans votre serveur, comportant deux phases :

1. attente du client,
2. traitement du client.

Le serveur ne doit pas fermer la socket d'écoute après la connexion d'un client, car il devra la réutiliser pour accueillir le client suivant. Il faudra aussi prévoir la terminaison du serveur, par exemple à la réception du signal `SIGTERM`. Lorsque vous envoyer ce signal au serveur, il doit terminer de gérer le client actuel, fermer sa socket d'écoute, et se terminer.

**1**) Commencez par ouvrir le code du serveur qui vous est fourni. Dans ce code, une fonction est appelée pour prendre en charge le dialogue du serveur avec un client : comme s'appelle-t-elle, et que reçoit-elle comme paramètre ? Est-ce que le fonction ferme bien la socket de dialogue une fois la connexion terminée ?

**2**) Modifiez le code `main()` de façon à ce que le serveur puisse traiter des clients arrivant successivement, comme cela a été décrit ci-dessus. Pour l'instant, ne vous préoccupez pas de la terminaison du serveur.

**3**) Pour tester votre code, lancez le serveur dans un terminal, puis connectez vous au serveur avec un premier client depuis un autre terminal. En laissant le premier client connecté, lancez un second client depuis encore un autre terminal. Que se passe-t-il pour le second client ? Quand parvient-il à se connecter ?

**4**) Lorsque le serveur reçoit le signal `SIGTERM`, quelle fonction de gestion du signal est appelée ? Que fait cette fonction ? Donnez une méthode pour envoyer `SIGTERM` à votre serveur.

**5**) En utilisant la variable booléenne `quit`, dont vous devez pressentir le rôle désormais, modifiez votre code de façon à ce que votre serveur se termine à la réception du signal `SIGTERM`. Précisons que, lorsqu'un processus reçoit un signal alors qu'il est en attente dans un appel système (c'est le cas dans `accept_connection()`), alors l'appel système se termine ; par contre, la fonction `dial()`) bloque la réception du signal jusqu'à ce quelle ait fini son exécution. Il vous suffit donc de consulter la valeur de `quit` après `accept_connection()` et après `dial()`. N'oubliez pas de fermer le socket d'écoute avant la terminaison du serveur.

**6**) N'oubliez pas de tester votre serveur : notamment, que se passe-t-il si vous lui envoyez le signal `SIGTERM` alors qu'un client est connecté ? Pourquoi ?

## EXERCICE 3 ► **Accueil des clients simultanément**

Dans cet exercice, vous allez repartir du code de l'exercice précédent, donc il n'est peut-être pas inutile de faire une sauvegarde de votre fichier `echoserver.cpp`.

On souhaite désormais que le serveur crée un processus fils à chaque fois qu'un nouveau client demande à se connecter. Chaque fils se charge du dialogue avec un client. Pendant ce temps, le serveur revient se mettre en attente sur la socket d'écoute. Vous serez confrontés aux petits tracas habituels lorsqu'un processus crée des fils multiples : pour éviter une accumulation de zombies digne d'un film d'horreur (et surtout pénalisante pour le système), on les éliminera après chaque nouvelle connexion.

**1**) Modifiez votre code de façon à ce qu'un processus fils soit créé lors de l'acceptation de chaque nouveau client. Le fils créé ferme la socket d'écoute (qu'il n'utilise pas), et prend en charge le client à l'aide la fonction `dial()` et de la socket de dialogue avec ce client.

**2**) Lancez votre serveur. Faites plusieurs connexions-déconnexions dessus avec votre client : vérifiez que votre serveur laisse bien traîner autant de zombies que vous avez fait de connexions-déconnexions ! Terminez votre serveur en lui envoyant `SIGTERM` avec la commande `kill` depuis la ligne de commande.

**3**) En consultant la page de manuel de `waitpid`, dites ce que fait l'appel `waitpid(-1, NULL, WNOHANG)`, et ce qu'il retourne. Une fois que vous aurez compris cet appel, vous pourrez mettre en place une solution pour que le père élimine tous les zombies qu'il a laissé traînés, et cela doit intervenir juste après la création d'un nouveau fils, et la mise en attente d'une nouvelle connexion. Une fois cette solution mise en œuvre, vérifiez par l'expérience qu'elle fonctionne !

**4**) Que se passe-t-il si vous envoyez le signal `SIGTERM` au serveur alors que des clients sont encore connectés ? Donnez votre réponse après avoir expérimenté !

**5**) On souhaite que le père attente la terminaison de tous ses fils avant de se terminer lors de la réception de `SIGTERM` : mettez cela en place dans votre serveur. A nouveau, vérifiez par l'expérience que votre solution fonctionne !