R. Carpa, S. Castellan et V. Villebonnet

Projet 2 : un *gestionnaire de données*

Modalités de rendu Le projet doit être rendu avant le **vendredi 20 mai à minuit**.

Une seule archive avec le nom *NOM_Prenom.zip* ou *NOM_Prenom.tar.gz* sera envoyée à vos encadrants de TD dans un e-mail avec le sujet "[ASR2] Rendu Projet2 de NOM Prenom". L'archive **doit** contenir un seul répertoire : *NOM_Prenom* lui même contenant vos sources. Le projet doit se compiler avec la commande *make* à la racine du répertoire.

Un fichier README.txt est impératif et doit, au moins, contenir quelques phrases expliquant quelles parties du sujet ont été réalisées ou non.

Dans ce projet, vous allez développer un gestionnaire de fichiers et de téléchargement de fichiers. Le serveur met à disposition un ensemble de fichiers qu'il est possible de télécharger par tous les clients qui s'y connecte. Il est également possible de demander au serveur de télécharger un fichier non déjà présent sur le serveur pour complèter la liste des fichiers proposés.

1 Architecture de base du serveur

Question 1

Écrivez un programme qui prend deux arguments : un nom de dossier et un port (entier). Vérifiez que les deux arguments passés sur la ligne de commande sont corrects.

Toutes les commandes opérant sur des fichiers seront relatives à ce dossier.

Question 2

Faites que votre programme soit un serveur qui écoute sur le port passé en second argument de la ligne de commande. À chaque nouvelle connexion, le serveur doit créer un nouveau *thread*, et lui donner la *socket* (descripteur de fichier) correspondant à la communication ouverte avec ce client. Pour l'instant, le *thread* va se contenter d'écrire *hello* au client et de fermer la *socket*.

Vous pouvez tester votre serveur avec l'utilitaire `nc` :

```
nc localhost <port choisi>
```

Si tout fonctionne bien, vous devez voir s'afficher simplement `hello` et la main vous est rendue.

Question 3

Écrivez une fonction `server_kill` qui itère sur la liste des threads clients, qui attend qu'ils terminent et ensuite ferme la socket serveur.

Question 4

Modifier la boucle du serveur qui accepte les clients pour écouter sur stdin en parallèle avec `select`. Si l'utilisateur tape `exit`, appelez `server_kill`.

2 Une boucle intéractive

Question 1

Dans le thread dédié au client, au lieu de refermer la socket tout de suite, faites une boucle :

- J'attends une commande du client de la forme `commande argument`

- J'exécute la fonction dédiée à la commande `commande` en lui passant l'argument.
- J'attends la commande suivante.

Les commandes sont de la forme :

```
<commande> <argument optionnel>
```

Et elles sont séparées par des retours à la ligne.

Les commandes sont des fonctions prenant un argument (optionnel) et possiblement un paramètre décrivant l'état du client, et renvoie un entier qui indique s'il faut garder la communication avec le client. Stockez-les dans un tableau comme cela était fait dans le projet 1.

Question 2

Faites en sorte de gérer de multiples commandes qui arrivent en même temps ou partiellement. Les exemples suivant doivent tous mener à l'exécution de la commande `foo` puis la commande `bar`:

```
(echo foo; echo bar) | nc localhost <port choisi>
(echo foo; echo -n ba; sleep 3; echo r) | nc localhost <port choisi>
```

Pour cela, il faut gérer vous-même votre tampon pour la réception de lignes incomplètes ou de plusieurs lignes en même temps.

3 Les commandes

Question 1

Implémentez une commande `quit` qui ferme la socket client.

Question 2

Implémentez une commande `clients` qui renvoie au client le nombre de clients connectés à cet instant. Pour cela, vous devez stocker les clients dans une liste (possiblement doublement) chaînée.

Question 3

Implémentez une commande `size <fichier>` qui renvoie au client l'entier correspondant à la taille du fichier `fichier`. Le chemin vers le fichier est relatif au nom de dossier qui a été passé en premier argument au lancement du serveur.

Question 4

Implémentez une commande `get <fichier>` qui envoie le contenu du fichier `fichier` au client.

Question 5

Implémentez une commande `list` qui renvoie la liste des fichiers disponibles (un par ligne), et termine par une ligne vide (pour signaler la fin de l'input).

4 Graceful shutdown

Un problème avec le serveur actuel est que pour terminer le serveur, il faut attendre que les clients se déconnectent d'eux-mêmes. On s'attaque maintenant au problème de déconnecter les clients en mode `idle` (ie. dont on est en attente d'une commande) lors d'un `server_kill`. On attend tout de même que les transferts (avec `get`) soit terminés avant de quitter.

Question 1

Dans le thread principal, ignorez les signaux `SIGINT` et `SIGUSR1`. Que se passe-t-il si vous appuyez sur `Control-C` dans le terminal du serveur?

Question 2

Dans chaque `thread`, utilisez la fonction `signalfd` pour binder `SIGUSR1` sur un descripteur de fichier. Écoutez sur cette socket en parallèle de la socket client pour détecter les envois de `SIGUSR1`.

Si vous recevez quelque chose sur ce descripteur virtuel, quittez la boucle et fermez la connexion avec `close`.

Question 3

Modifier la fonction `server_kill` qui itère sur la liste des threads clients, pour leur envoyer `SIGUSR1` avant d'attendre qu'ils terminent.

Question 4

Faites en sorte que cette fonction soit appelée lors de la réception d'un signal (appel à select interrompu dans le thread principal) et lorsque l'utilisateur tape `exit` dans le terminal.

Question 5

Implémentez une commande `shutdown` qui envoie un `SIGINT` au thread principal s'assurant ainsi que le serveur soit bien terminé.