TP 4 : Multi-clients : Pool de thread

Le but de ce TP est de voir comment gérer plusieurs clients simultanément. Le CR est à rendre sur AMETICE avant la prochaine séance.

Serveur simple

Reprendre le serveur de TP précédent, expliquer *précisément* ce qui se passe lorsque deux clients se connectent en même temps au serveur.

Solutions ?

Quelles solutions connaissez-vous pour gérer les entrées/sorties concurrentes et asynchrones ?

Thread

Requête

Modifier le serveur afin de créer et utiliser un thread pour gérer chaque nouveau client.

Test de performance 1

Ecrire un programme `Stress1.java` qui prend comme paramètre un entier, connecte `$n$` *clients stressants*. Un client stressant est une méthode qui ouvre une socket client et envoie une seule ligne `client stress1 n°i` sans fermer la connexion ensuite.

Comparer le comportement de votre serveur pour les valeurs de `$n$` :

- 1
- 2
- 10
- 100
- 1 000
- 5 000

Modifier le programme `Stress1` pour fermer les connexions immédiatement.

Y a-t-il une différence de comportement de votre serveur pour les mêmes nombres de connexion entre les clients `Stress1` selon que vous fermez la connexion immédiatement ou non ?

Que se passe-t-il si vous lancez 5 clients `Stress1` simultanément avec comme paramètre 1 000 ?

Performance temporelle

Afin de mesurer le temps de réponse du serveur (à des requêtes type fixées), nous allons ajouter des appels à l'horloge via `System.nanoTime()`.

Pour chaque valeur de n mesurer le temps de réponse et préparer un graphique à partir de ces mesures. (**Indication :** On pourra exporter un fichier `.csv` (champs séparés par `;`) pour collecter les mesures).

Optionnel : Ecrire un programme `Stress2.java` qui prend comme paramètre un entier, se connecte en envoyant deux lignes `client stress n°i`, sans fermer la connexion ensuite, et mesure le temps de réponse à la première ligne, à la seconde.

Pool de Threads

Afin d'améliorer les performances de votre serveur, nous allons utiliser la technique du *pool de threads*, qui permet de réutiliser les threads et donc limite la consommation de ressources liée à la création du thread.

Il existe différent types de pool :

- les pools dynamiques
- les pools statiques
- les pools "voleurs"

Syntaxiquement, le principe est le même et tous sont regroupés dans la classe ``Executors``. Ces premiers pools sont créés avec `Executors.newCachedThreadPool`. Il existe deux signatures pour cette méthode : quelles sont les différences entre les deux ? Les deuxièmes sont créés avec avec `Executors.newFixedThreadPool`. Existe-t-il plusieurs signature pour cet appel ? Pourquoi ? Le dernier cas correspond à des pools de taille fixe qui vise à s'ajuster au niveau du parallélisme "réel" du système.

Serveur avec pool dynamique

Modifier le code de votre serveur afin créer un pool de thread.

On modifiera également le traitement de la ligne de commande par le programme afin de respecter la syntaxe d'usage suivante :

```
Usage: java EchoServerDPool [-t nb] port


port : n° de port utilisé par le serveur

nb (optionnel) : nombre de threads associés au serveur,

                 si non défini alors le pool est dynamique
```

Test de performance 2

Reprendre le client `Stress.java` et tester pour les valeurs de n :

- 1
- 2
- 10
- 100
- 1 000
- 5 000

Que constatez-vous ?

Serveur avec pool statique

Modifier le code de votre serveur afin créer un pool de thread de taille fixe.

On modifiera également le traitement de la ligne de commande par le programme afin de respecter la syntaxe d'usage suivante :

```
Usage: java EchoServerSPool -t nb port


port : n° de port utilisé par le serveur

nb : nombre de threads associés au serveur
```

Test de performance 3

Reprendre le client `Stress1.java` et tester pour les même valeurs de `$n$` :

Que constatez-vous ?

Serveur avec pool "voleur"

Ecrire un nouveau serveur qui utilise `newWorkStealingPool()`.

Test de performance 4

Reprendre le client `Stress1.java` et tester pour les même valeurs de `$n$` :

Que constatez-vous ?

Bilan

Mettre sous forme de graphiques les différentes mesures obtenus pour chacun des types de pool. Conclure sur les différences de performances.

Outils

Veuiller noter les commandes suivantes permettant de contrôler les serveurs lancés automatiquement :

- pour voir les applications java en cours d'exécution : `$ jps -v`
- pour voir les threads d'un processus java (attention, c'est verbeux!) : `$ jstack $pid`
- pour voir le pid d'un processus lorsqu'il démarre : `$ ./programme & fg`
- pour récupérer le pid d'un processus qui vient d'être lancé : `$ pid=$!`
- pour voir les threads linux d'un processus : `$ ps -Tf $pid`
- pour voir les limites de ressources d'un processus : `$ cat /proc/$pid/limits`