Marc-Antoine Beaulieu BEAM19119007

Martin Desharnais DESM21099102

LOG735 Système distribués

Laboratoire 1

Synchronisation et concurrence dans les systèmes distribués

Travail présenté à :

Monsieur Thierry Blais-Brossard

École de technologie supérieure

20 mai 2015

Introduction

Ce laboratoire consiste à apporter des améliorations successives à un système client-serveur de type ECHO en majuscule. D'un code de base où le serveur n'accepte qu'une connexion, nous ajoutons la possibilité à plusieurs clients de se connecter. Nous ajoutons ensuite une tolérance aux fautes en nous connectant automatiquement à un autre serveur après dix secondes sans réponse. Nous ajoutons ensuite un compteur global du nombre de requêtes traité sur le serveur. Nous ajoutons ensuite une synchronisation entre les différents serveurs afin que le compteur soit partagé.

Q1

Tel qu'écrit, le serveur n'accepte la connexion que d'un seul client. Le premier à tenter de se connecter réussira probablement à établir la connexion et à utiliser le service du serveur. Les clients subséquents n'auront pas cette chance. Il n'y a aucune gestion du multithreading.

Q2

La fonction main du client a été modifiée afin de corriger le défaut qui l'empêchait de se terminer lorsque la connexion avec le serveur était interrompue (e.g. avec la commande « Bye. »). La fonction main du serveur a été modifiée afin que la connexion de plusieurs clients soit acceptée dans une boucle infinie. Chaque fois qu'un client se connecte, une nouvelle tâche est ajoutée dans un thread pool afin de traiter la demande de ce client.

Q3

Nous avons extrait le code responsable de se connecter au serveur dans une fonction « getSocket ». Celle-ci prend en argument les informations de connexion (i.e. adresse IP et numéro de port) et retourne un socket ouvert. Nous en profitons pour définir un minuteur de 10 secondes après lesquelles le socket lancera une exception « SocketTimeoutException ».

Le code principal utilise cette fonction pour se connecter au premier serveur de la liste. Dans le cas où une exception de minuteur serait lancée, le code se connecte au serveur suivant dans la liste et recommence là où il était rendu. Pour ce faire, nous avons dû placer l'instruction de l'utilisateur dans une variable afin de pouvoir poursuivre sur la même instruction en cas de changement de serveur.

Le code principal a été placé dans une boucle qui continue tant que la tâche n'est pas terminée et qu'il reste des serveurs auxquels tenter de se connecter.

La liste des serveurs est simplement passée au programme par les arguments de la fonction « main » sous la forme « ip:port » (e.g. client 127.0.0.1:1234 127.0.0.1:1234). Ce choix a été motivé par la simplicité d'implémentation (i.e. il est très simple d'avoir accès aux arguments du programme), la simplicité d'utilisation (i.e. il est très simple de fournir des arguments à un exécutable) et l'extensibilité (i.e. un nombre arbitraire de serveurs peut être fourni).

De mettre les coordonnées en paramètre offre une certaine transparence d'accès, dans le sens où un

serveur local peut être accédé de la même manière qu'un serveur distant. De plus, ce choix a eu un grand impact sur la transparence de localisation : il est nécessaire de connaitre les adresses des services pour s'y connecter. Par contre, notre choix n'a aucun impact sur la transparence de défaillance, de mobilité, d'extensibilité ou de performance; que les coordonnées soient dans le programme ou passé en argument, le programme demeure autant transparent d'une manière que de l'autre.

Q4

Non, il n'a pas été nécessaire de synchroniser les serveurs.

Q5

Un compteur atomique global a été ajouté à la fonction « main ». Le compteur susnommé est injecté dans les tâches et celles-ci l'utilisent afin d'afficher le nombre de requêtes total traité par le serveur. Ceci est fait dans la méthode « EchoTask.run ».

Q6

La méthode « main » a été modifiée afin d'ouvrir un socket écoutant en multicast et de lancer une tâche en arrière-plan (« CounterSynchronizationReceiveTask ») afin de mettre à jour le compteur. Chaque fois que le serveur traite une requête, une tâche de synchronisation (« CounterSynchronizationSendTask.java ») des autres serveurs est lancée.

Q7

Principalement, deux tâches de synchronisation ont été ajoutées :
« CounterSynchronizationReceiveTask », qui se charge de recevoir les messages des autres serveurs pour incrémenter le nombre de requêtes reçues et « CounterSynchronizationSendTask », qui a pour but d'envoyer aux autres serveurs le nombre actuel de requêtes lors de leur réception.

La synchronisation s'effectue sur un socket UDP multicast entre les serveurs sur le même groupe. Une charge utile de 8 octets est envoyée sur le socket lors de la réception d'une requête par un serveur. Les quatre premiers octets identifient le serveur qui envoie le paquet et les quatre derniers contiennent le nombre de requêtes reçues. Le ou les serveurs qui reçoivent le paquet mettent à jour leur compteur local avec ce nombre.

Des erreurs de synchronisation pourraient se produire, avec cette technique, si un paquet UDP ne se rend pas à destination. Ce problème pourrait être corrigé avec des protocoles plus élaborés qui gèrent la garantie de réception, par exemple. Aussi, bien que l'exercice ne le demande pas, si deux serveurs reçoivent une requête d'un client avant de recevoir le message de synchronisation avant l'envoi de leur réponse, une incrémentation pourrait être évitée et causer une faute.

Q8

Deux concepts fondamentaux furent abordés dans ce laboratoire : la tolérance aux fautes ainsi que

la synchronisation entre les systèmes.

La tolérance aux fautes consiste à ce qu'un système soit capable de détecter qu'une faute à lieu et de prendre des actions correctives afin d'en diminuer l'impact. Dans ce cas-ci, dès lors qu'un serveur ne répondait pas pendant dix secondes, la connexion à celui-ci était abandonnée, un nouveau serveur était contacté et les opérations recommençaient là où elles avaient été laissées. Le tout se faisait sans intervention de l'utilisateur.

La synchronisation entre les systèmes consiste à ce que des informations importantes soient synchronisées. Dans ce cas-ci, il s'agissait du compteur global. La technique que nous avons choisie est de diffuser l'information à chaque fois qu'elle est modifiée. Une autre technique aurait pu être de la diffusé à toutes les x secondes.

Dans l'éventualité où nous voudrions ajouter des serveurs de relève supplémentaires, il suffirait d'en informer les clients qui l'ajouteraient aux arguments du programme.

Conclusion

En conclusion, ce laboratoire nous a permis d'expérimenter avec les concepts de tolérance aux fautes et de synchronisation entre serveurs. Nous avons démontré qu'il est relativement facile et rapide de mettre en place des implémentations simplistes de telles fonctionnalités. Cependant, des techniques plus avancées seraient nécessaires afin de dépasser le stade de simples prototypes.