Systèmes et Réseaux

Réseaux de Kahn

Auteurs :
Nicolas ASSOUAD
Ismail LAHKIM BENNANI

26 mai 2016

Introduction

Dans le cadre du cours "Systèmes et Réseaux" dirigé par le professeur Marc Pouzet, nous avons été amenés à implémenter de différentes manières des réseaux de Kahn. Un réseau de Kahn est un modèle d'exécution parallèle de processus déterministe. Ces réseaux sont déterministes, c'est à dire que peut importe la manière dont il sera exécuté, le résultat sera toujours le même. Une autre caractéristique des réseaux de Kahn est que le seul moyen de communication entre les processus sont des files de longueur théorique infinie.

Nous avons réalisé trois implémentations différentes de ces réseaux de Kahn : une séquentielle, une avec des processus, et une en réseau.

1 Implémentation avec des processus

Nous avons réalisé une version avec des processus. Les channels sont représentées par des pipes. Le doco revient à effectuer un certain nombre de fork pour exécuter les différentes fonctions en parallèle sur des processus fils.

2 Implémentations séquentielles

Nous avons réalisé une version à exécution séquentielle (parallélisme simulé). Toute l'idée de cette implémentation est de savoir lors de l'exécution d'un doco comment jongler entre les exécutions des différentes fonctions en parallèle. Nous avons eu deux approches dans cette implémentation.

La première qui correspond au fichier "sequence.ml" prend le parti de changer de fonction qui s'exécute seulement lors de get bloquant, c'est à dire de get vers une channel vide, ou lors d'un put, ce qui correspond à une mise à jour d'une channel. Cette première approche n'est pas dénuée de sens, en effet, les changements d'exécutions ne font que lorsqu'ils sont vraiment utiles : une fonction passe la main seulement lorsqu'elle est bloquée par un get, ou lorsqu'elle a effectué un put qui peut favorable à l'exécution d'une autre fonction.

La seconde approche qui correspond au fichier "sequence_ord.ml" prend un parti plus extrême : on n'exécute qu'un pas élémentaire de chaque fonction à la fois. Cela se matérialise de le code dans la fonction bind : celle-ci ne calcule plus les deux fonctions qu'elle possède en argument, mais seulement la première et "renvoie" la deuxième. Ainsi dans cette approche, chaque fonction d'un doco avance de la même manière.

Concrêtement dans les deux approches, les interruptions du calcul des fonctions est effectuées par des exceptions. Ces exceptions contiennent la suite du calcul à

effectuer, c'est à dire la continuation, ou bien dans le cas d'un get bloquant le même get réitéré. Ces calculs sont gelés dans l'exception à l'intérieur d'une fonction prenant unit pour argument.

3 Implémentations en réseau

L'implémentation en réseau a été la plus périlleuse de toute. Notre but été le suivant : avoir une implémentation qui puisse à la fois marcher sur un ordinateur local, sur plusieurs machines différentes comprenant ou non la machine ayant lancé le réseau de Kahn.

La grande difficulté rencontrée a tournée autour de la marshalisation. En effet, le comportement de la marshalisation en présence de descripteur de fichier, notamment de pipe et de socket, n'est pas très bon. Aucun descripteur de fichier ne doit être présent dans la portée de la fonction à marshaliser lors de la marshalisation (en cela, je pense particulièrement à des variables globales qui auraient pu être bien pratique). Toute la difficulté de ce constat vient donc de l'impossibilité de posséder un moyen de communication par pipe entre les fonctions marshalisées typiquement des get, des put, et des new_channel et les autres composantes du programme.

Cette difficulté exposée, passons à la strucure de notre solution. Notre implémentation repose sur des serveurs d'exécutions qui seront présents sur les différentes machines qui effectueront les calculs. Ces serveurs sont composés de deux processus principaux :

— une entité chargée de récupérer tout ce qui arrive sur le port du serveur et de le rediriger vers un pipe local. La fonction en charge de cette tâche est la fonction "network_buffer". Elle utilise la fonction de haut niveau establish_server qui fork un nouveau processus charger de la redirection pour chaque connexion entrante.
— une entité chargée d'interpréter les différentes requêtes. Il lit des informations sur le pipe local alimenté par le "network_buffer" et y répond de manière appropriée. La fonction en charge de cette tâche est la fonction "request_manager". Chaque requête est composée du nom de la requête, d'une en tête contenant les informations d'identification de la requête qui sont l'identifiant du processus qui a envoyé la requête (son PID) et de l'adresse où il se trouve sur le réseau, et de données spécifique à la dite requête. Les différentes requêtes qui peuvent être rencontrées sont :

    1. "FORK", cette requête reçoit une fonction marshalisée et va la faire exécuter sur un processus fils. C'est ici qu'intervient la difficulté décrite au dessus : il nous faut un moyen de communication entre le processus nouvellement crée et le request_manager. Pour chaque processus, nous allons crée un pipe nommé. Son nom sera composé du nom de la machine et de son PID. Une autre particularité est que le request_manager s'arrête pas

pour attendre son fils, il a d'autre requête à gérer. Nous avons donc mis en place la technique du double fork pour éviter tout processus zombie.

2. "NEW_CHANNEL", cette requête crée une nouvelle channel. Les channels comme dans l'implémentation procédurale sont représentées par des pipes auxquels on ajoute un compteur qui garde en mémoire le nombre d'élements présent à l'intérieur du pipe.
3. "PUT", cette requête recoit l'identifiant de la channel et la valeur à y mettre. Il vérifie qu'il y a suffisament de place dans la channel grace au compteur, si c'est le cas il y incorpore la valeur, sinon il remet une requête PUT identique sur la pile des requêtes à gérer.
4. "GET", cette requête recoit l'identifiant de la channel dans laquelle il faut lire. Il vérifie qu'il y a bien des élements à lire (la lecture étant bloquante si le pipe est vide), si c'est le cas il les lit, sinon il remet une requête GET identique sur la pile des requêtes à gérer.
5. "TERMINE", cette requête ne prend pas d'en tête, mais seulement l'identifiant d'un processus et une valeur. Elle a pour rôle d'envoyer via le pipe nommé une valeur de retour au processus identifié par l'identifiant reçu.

Toute les requêtes exceptées la requête TERMINE se termine par l'envoie d'une requête TERMINE aux serveur indiqué par l'adresse reçu dans l'en tête. L'identifiant passé est le même que celui présent dans l'en tête et la valeur dépend de la requête.

Les fonctions du réseau de Kahn peuvent s'exécuter de deux manières différentes (cela se matérialise dans le code par le code de la fonction bind, ou bien l'appel du processus dans la requête FORK) :

— Soit la fonction se termine et renvoie une valeur, c'est le meilleur cas
— Soit la fonction lève l'exception Exit comme typiquement dans les fonctions get, put, ou new_channel, alors le processus doit attendre la valeur de retour qui doit lui parvenir par le pipe nommé à son nom.

Le dernier point de compréhension de notre code sont les fonctions du réseau de Kahn. Les fonctions get, put et new_channel n'effectuent que des connexions au serveur local pour envoyer la requête adéquate avec les bonnes informations.

Notre implémentation rencontre un léger problème de concurrence du au fonctionnement des fonctions get et put. En effet, ces fonctions effectuent des connexions avec le serveur local par l'intermédiare de socket. Le problème est que l'ouverture d'une connexion ne peut être effectuée q'une seule fois simultanément sur un même port. Lors de l'exécution concurrente d'un get et d'un put, ce phénomène peut provoquer un blocage d'un get qui a déjà ouvert sa connexion mais pas fini sa transaction par un put mis en activité par l'ordonnanceur qui tenterait d'ouvrir une connexion déjà ouverte par le get. Cela bloque momentanément l'exécution jusqu'à ce que le get soit réordonnancé en premier plan pour qu'il termine son exécution.