Systèmes et réseaux
Projet Réseaux de Kahn

Axel Davy
École normale supérieure

Baptiste Lefebvre
École normale supérieure

26 mai 2013

1 Installation

Une fois l'archive `davy-lefebvre.tgz` récupérée et les fichiers extraits placez-vous dans le répertoire `davy-lefebvre`. Dans ce répertoire la commande :

- `make` : compile notre programme

Pour la désinstallation la commande :

- `make clean` : efface tous les fichier que la commande précédente a pu engendrer et ne laisse dans le répertoire que les fichiers sources

Si vous n'avez pas réussi à récupérer l'archive vous pouvez également récupérer le code source à l'aide de la commande

- `git clone https://github.com/KahnProcessNetworks/KahnProcessNetworks`

en vérifiant à bien avoir installé au préalable le système de contrôle de version `git`.
Le dossier Trash contient quelques fichiers correspondant aux premières implémentations que l'on avait réalisé pour le réseau et la simulation séquentiel.
Le dossier Test contient quelques fichiers tests. Par défaut la commande `make` compile le programme dont le code source est contenu dans le fichier `test.ml`.
Comme demandé, le fichier `kahn.ml` contient les modules `Th`, `Pipe`, `Socket`, `Sequential` et `Best`.
L'utilisation de l'implémentation `Socket` nécessite une procédure particulière pour lancer votre programme après l'avoir compilé (cf. partie 2.2 Implémentation Network).

2 Commentaires

2.1 Implémentation Pipe

Pas de commentaires particuliers.

2.2 Implémentation Network

La toute première implémentation de Network que l'on a réalisé était assez peu performante : on avait considéré que les `put` et les `get` étaient des requêtes à un serveur représentant le channel. Le problème de performance était au niveau de l'implémentation : chaque `put` et `get` utilisait `connect` pour se connecter au

serveur, puis une fois le `put` et `get` terminé, on fermait le socket. Des problèmes sont apparus dûs au fait que l'on réalisait un grand nombre de connections par secondes. Les problèmes ont été résolus en paramétrant les options des sockets, mais l'implémentation était vraiment lente.

Finalement au lieu d'améliorer cette implémentation en gardant les connections actives, et donc en évitant de refaire des connections à chaque `put` et `get`, nous sommes parti sur un tout autre concept permettant d'utiliser vraiment plusieurs machines du réseau, au lieu d'utiliser le réseau local de notre ordinateur.

Nous avons cherché à faire une implémentation vraiment complète à ce niveau : au départ plusieurs machines sont en attente d'ordres et une machine commence le programme. Chaque `doco` fait appel à des machines en attentes pour réaliser les taches et parallèlement met la machine ayant fait le `doco` en attente elle-même. Les machines ayant reçu un ordre font un fork pour avoir une instance du programme qui attend des ordres et une autre qui exécute, ce qui fait que toutes les machines sont susceptibles à tout moment de recevoir des ordres.

De manière plus détaillée, pour lancer le programme toutes les machines sauf une exécute le programme avec l'option `-wait`. L'autre machine en question, que l'on nommera mère, commence l'exécution. Chaque machine possède en local un fichier de configuration `network.config` qui contient l'ensemble des machines auxquelles elle est susceptible de demander l'exécution d'un processus. De plus la machine mère posséde un fichier de configuration `host.config` qui contient son hostname complet (DNS).

La machine mère commence par un fork qui lui permet d'établir un serveur disponible pour de futures demandes d'exécutions de processus de la part du réseau. De l'autre côté elle exécute le processus séquentiellement jusqu'à atteindre le premier `doco`. Ce qui suit est désormais également valable pour toute machine du réseau qui aura à traiter un `doco`. Il faut à la fois distribuer les processus à travers le réseau et établir les connections entre ces processus. La distribution est facile et chaque processus se retrouve associés à une machine. Les channels lorqu'ils sont créés sont identifiés de manière symbolique : ce sera uniquement lors d'un appel à `get` ou `put` que la connection se fera réellement. Pour établir cette connection l'identification des machines aux extrémités est nécessaire, elle se fera par une requête à la machine ayant lancé le `doco`. Cette machine doit donc prévoir un serveur pour ce service. Pour résumer, un `doco` en parallèle lance ce serveur et envoit les processus à travers le réseau puis attend leurs terminaisons. Afin de rendre indépendants les `get` et les `put` des aléas du réseau et des appels bloquants, un processus relais est créé pour accumuler les paquets et les redistribuer à la demande.

Nous avons rencontré un certain nombre de problèmes pour cette implémentation dont le plus gros était l'utilisation du module Marshal qui ne supportait pas les types abstraits (que l'on utilisait pour définir les connections ouvertes). Un autre problème a été celui de la gestion des signaux d'interruption. Après avoir essayé de faire des retours sur pannes, nous avons finalement opté pour une simple accumulation des identifiants des processus pour tous les tuer en cas de besoins. En ce qui concerne le débugage nous n'avons pas réussi à trouver d'autre moyen que d'effectuer des rapports de logs sur les différentes machines. Pour finir les serveurs utilisent un fork pour pouvoir servir un client tout en restant disponible à tout connexion entrante. L'utilisation de double fork serait

une optimisation possible.
guide d'utilisation :

- Spécifiez l'utilisation du module `Socket` dans `test.ml`
- Compilez avec `make`
- Copiez l'exécutable sur un ensemble de machines qui constituent le réseau
- Configurez les fichiers `network.config` pour chaque machine, chaque ligne doit correspondre à un host potentiel
- Configurez le fichier `host.config` pour la machine mère uniquement en y mettant son DNS
- Lancez le programme avec l'option `-wait` sur toutes les machines sauf la machine mère
- Lancez le programme sur la machine mère

Remarque importante :

- Il faut veiller à ce que toute machine mentionnée dans un `network.config` exécute le programme avec l'option `-wait` afin de pouvoir recevoir et exécuter un processus

2.3 Implémentation Sequential

Notre première approche a été de considerer qu'un processus pouvait soit terminer, soit rendre la main sans avoir terminé. Dans cette implémentation, les processus avaient le type `'a process = ('a -> unit) -> Status` avec `Status` indiquant si le processus avait terminé, ou s'il fallait le rappeler. `doco` devait donc appeler un à un les processus et en fonction du résultat renvoyé, savait s'il fallait rappeler le processus plus tard ou pas. Le problème de cette implémentation se situait au niveau de `bind` : `bind e e'` devait au premier appel exécuter le processus `e` et conserver son résultat, puis au second appel exécuter `e'` avec le résultat conservé, mais surtout aux appels suivant conserver les modifications appliquées à `e'` (qui à chaque appel pouvait évoluer). Par exemple si l'on execute le processus `integers` de l'exemple donné, après n appels au processus, on se retrouve à n appels de `bind` imbriqués. La complexité de l'implémentation était en $O(n^2)$, ce qui n'était pas satisfaisant.

Pour améliorer l'implémentation, le retour des fonctions a été remplacé par une exception à lever si le processus devait être rappelé. Le programme s'exécutait plus rapidement (un facteur 3 environ), mais la complexité était toujours en $O(n^2)$.

Finalement cette dernière implémentation a été améliorée pour renvoyer lors de l'exception la fonction permettant d'exécuter la suite du processus. `doco` faisait ensuite appel à cette fonction et non pas au processus initial qui avait été modifié. Il n'y a donc plus le problème d'appels imbriqués et la complexité est linéaire.

Lors de nos tests, la vitesse de cette implémentation était très proche de celle de pipe.