Rapport de projet : Réseaux de Kahn

I.Implémentations Unix et séquentielles.

L'implémentation Unix diffère de l'exemple donné par 2 critères : les channels ont été remplacés par des pipes, et le thread sont remplacés par des fork.
Sur tous les autres points, cette implémentation est similaire à l'exemple donné.

Pour ce qui est de l'implémentation séquentielle, j'ai remplacé le type **('a → unit) → unit** proposé pour implémenter les processus par le type suivant :

```
type 'a process = | End of 'a
                  | Compute of (unit → 'a process)
```

Ce type à l'avantage de représenter intuitivement les étapes de calcul. De plus, il masque l'implémentation de la file de processus à exécuter par une définition astucieuse de la fonction doco. L'idée vient de Paul-Nicolas Madeleine (Info16), mais j'ai moi-même réalisé les implémentations.

II. Implémentation réseau

Le plus grosse partie du projet consiste à proposer une implémentation réseau des réseaux de Kahn. Pour cette dernière, le modèle utilisé est le suivant :

Il s'agit d'un modèle centralisé, dans lequel la commande est lancée depuis le serveur, qui redistribue ensuite les taches entre les différents clients connectés. Ceux-ci exécutent les différents processus en parallèle. Tout échange entre un processus et le serveur passe d'abord par le client concerné.

Un processus est représenté par la type **unit → 'a** décoré d'information supplémentaires, qui servent à l'implémentation pratique de certaines fonctions (telles que doco).

Les channels sont stockés sur le serveur, et chaque échange de donnée doit donc passer par celui-ci. Ils peuvent donc être identifiés de manière unique par un identifiant.
Un des défaut de cette méthode est la difficulté à implémenter des conditions pour bloquer un processus voulant envoyer des données (l'exemple donné affichant les entiers par exemple, génère une *segmentation fault* au bout de quelque secondes si l'utilisateur n'a pas pris soin de rajouter lui-même des temps d'attente).
D'une manière générale, il est difficile d'équilibrer automatiquement la taille des channels dans l'implémentation actuelle (cf partie **III**).

La fonction new_channel a aussi changé de signature, et renvoie désormais un processus qu'il faut bind pour des raisons pratiques d'implémentation. Cela permet de demander un nombre ilimité de channels par client, et ne complexifie pas beaucoup la création de processus. Ce choix de programmation est à comparer à celui de Corentin Barloy, qui a gardé la même signature au prix d'un nombre limité de nouveaux channels par client. Son implémentation est aussi plus rapide sur ce point, puisqu'elle ne nécessite pas de communication avec le serveur pour définir un nouveau channel.

Enfin, mon implémentation comprend une fonctionnalité permettant de déconnecter un client alors que le calcul global n'est pas terminé. Elle n'est que peu utile pour le moment, mais les

implémentations futures permettront de la rendre moins contraignante. Ce point est abordé plus en détail ci-dessous.

III. Implémentations à venir

Actuellement, plusieurs problèmes rendent ce projet incomplet, et constituent donc des pistes d'implémentation pour le futur.

Tout d'abord, l'interface servant à la déconnexion est parfois inutilisable lorsqu'un ou plusieurs processus font de nombreux affichages à cause de la fusion dans la konsole de stdin et stdout.
Une meilleure implémentation consisterait à traiter le signal d'arrêt forcé du processus (Ctrl+C) pour remplacer l'input « \q » demandée pour initialiser la déconnexion.

Un autre problème survient lors de la déconnexion d'un client. Souvent, il contient des processus de la forme **doco l** qui ne s'arrêtent qu'à la fin de l'exécution du programme, et rendent la déconnexion prématurée inutile. Il faudrait donc implémenter une version volatile de ces processus pouvant être renvoyés à tout moment par le client pour initier sa déconnexion.
Une solution serait de remplacer le type **unit → 'a** par un type **(unit → 'a) list**, où l'on à coupé le processus initial après chaque **doco**. Ainsi, le client pourrait renvoyer les processus en attente de terminaison d'un **doco** si cela est souhaitable.

Enfin, le problème de la gestion de la taille des piles sur le serveur reste le facteur majeur de crash de cette implémentation. Il est donc nécessaire d'implémenter un garde-fou à l'envoi de données pour garantir la sureté de l'application.

Le code à jour des différentes implémentations des réseaux de Kahn évoquées ci-dessus est disponible à l'adresse suivante : https://github.com/astraxel/Vanille