PRR-Labo2

Tiago Povoa et Burgener François

Donnée du laboratoire

Objectifs

- Comprendre le fonctionnement d'un algorithme d'exclusion mutuelle réparti.
- Utiliser l'algorithme de Carvalho et Roucairol pour maintenir la cohérence d'une variable répartie sur un ensemble de processus.
- Réaliser des communications TCP en langage Go

Énnoncé du problème

> Partager une donnée parmi un ensemble de processus est un problème qui peut se résoudre par le biais d'un algorithme d'exclusion mutuelle. Dans ce laboratoire, nous allons utiliser l'algorithme de Carvalho et Roucairol, une optimisation de l'algorithme de Ricart et Agrawala, comme algorithme d'exclusion mutuelle. Chaque processus détient une variable entière qui doit être cohérente. Les tâches applicatives peuvent faire 2 opérations sur cette variable : consulter sa valeur, et modifier sa valeur. La consultation revient à obtenir la valeur la plus récente. Par contre, une modification se passe en 3 étapes :
>
> 1. obtenir l'exclusion mutuelle sur la variable ;
> 2. modifier la valeur en section critique, par exemple l'incrémenter ;
> 3. informer tous les autres processus de la nouvelle valeur ;
> 4. libérer la section critique.

Comment démarrer

Nous avons 3 manières de lancer notre projet. Via deux script, windows et linux, ou alors via ligne de commande

Windows

Pour lancer le script il faut aller dans le dossier labo2 `PRR-Labo2/labo2` et de lancer le script `startWindows.bat`

```
$ ./startWindows.bat <nombre de processus>
...
Appuyez sur une touche pour continuer...
```

L'argument est le nombre de processus que l'on souhaite lancer. Ensuite appuyer sur une touche pour qu'un processus se lancer. Une fois que le processus a terminé de exécuter appuyer de-nouveau sur une touche pour lancer le prochain processus. Fait cela pour tout les processus.

Linux

Pour utiliser le script sur linux, il vous faut avoir le terminal gnome-terminal. Si vous ne l'avais pas, vous pourrait lancer chacun des processus via ligne de commande.

Pour lancer le script il faut aller dans le dossier labo2 `PRR-Labo2/labo2` et de lancer le script `startLinux.sh`

```
$ ./startLinux.sh <nombre de processus>
```

L'argument est le nombre de processus que l'on souhaite lancer.

Ligne de commande

Pour lancer en ligne de commande il faudra tout d'abords aller dans le dossier `PRR-Labo2/labo2` et exécuter la ligne suivante dans différent terminal

```
go run main.go -proc <id du processus> -N <nombre de processus>
```

Les id des processus commencent à **0**

Example

```
go run main.go -proc 0 -N 3
go run main.go -proc 1 -N 3
go run main.go -proc 2 -N 3
```

Prise en main

Il est possible d'activer un mode debug dans `processus` pour le mutex OU le network (au choix). Ceci facilitera l'analyse de l'exécution.

Travail réalisé

Afin de réaliser un meilleur découpage de notre code que le laboratoire précédent, nous avons répartit en deux struct notre code. Afin d'éviter de polluer l'espace global avec des chanels innombrables, nous avons encapsulés ceux-ci à l'intérieur de méthodes afin de fournir une API concise et clair.

Mutex

Api disponible

Client

Méthode	
Init	"Constructeur": initialise le mutex et ses valeurs internes
Ask	Prépare l'entrée en section critique en démarrant les appels aux autres mutex
Wait	Attente bloquante sur la disponibilité de la ressource critique
End	Libère la ressource critique
GetResource	Obtient la valeur de la ressource critique
Update	Permet de mettre à jour la valeur

Réseau

Méthode	
Req	Accepte une requête (depuis le réseau)
Ok	Accepte un "OK"
Update	Accepte un update

En détails

Lorsque l'on va initier notre Mutex, une goroutine va se démarrer avec la méthode `manager`. Cette méthode attend sur la réception de chanels et traite les demande entrantes.

Cas échéant, il est capable de transmettre et déléguer à un objet implémentant l'interface `Networker` ce qui doit se faire sur le réseau.

Network

Api disponible

Méthode	
Init	Initialise le network. Prépare des dials et des accept => À la fin on se retrouve avec une (et une seule) connexion par pair de noeuds
REQ	Transmet une requête à un autre processus réseau
OK	Transmet un OK à un autre processus réseau
UPDATE	Transmet une mise à jour de la valeur protégée en section critique à tous les processus

En détails

Lors de l'initialisation, nous devons établir les connexions TCP entre chaque noeud. On va commencer par faire une boucle de Dial sur chaque autre processus distant. Dans le cas d'un échec, la connexion sera établie à posteriori via un Accept. Chaque connexion est stockée dans `directory` qui est une map de `net.Conn`.

Une fois cette première étape terminée, notre partie client s'exécute.

Lorsque c'est nécessaire, il délégue les opérations à un objet implémentant l'interface Mutex.

protocole

Message	Format	Taille
Req	[REQ {Stamp} {Id}]	3 + 4 + 2 bytes
Ok	[OK_ {Stamp} {Id}]	3 + 4 + 2 bytes
Update	[UPD {value}]	3 + 4 bytes

Les messages sont terminés par des `\n`

Autre

Le point d'entrée `main.go` représente notre client. Dans notre partie client, nous avons un simple programme en lignes de commandes qui accepte trois commandes:

1. Lire
2. Update
    1. Entréer une valeur numérique naturelle
3. Quitter

Le package `processus` contient une struct qui permet d'initialiser facielement le mutex et le network (étant donné qu'ils ont des valeurs communes).

Le package `config` permet de configurer certains aspects du programme (constantes).

Améliorations

- Actuellement, notre partie network ne nécessite pas un ordre particulier. En revanche, si on lance en simultané les processus, il est possible qu'ils se bloquent les deux en Accept. Nous sommes contraints de les démarrer séquentiellement, peu importe l'ordre.
- Actuellement, nous effectuons des logs selon une variable Debug séparament entre Mutex et Network. Nous pourrions rediriger le stdout vers un fichier ou autre chose afin de faciliter la lisibilité.