Projet systèmes concurrents/intergiciels

2IR

23 février 2024

1 Canaux Go

L'objectif du projet est de réaliser une implantation de la communication par canaux synchrones, tels qu'ils existent en Go ou en π-calcul.

Un canal (interface `go.Channel`) possède deux opérations `in()` et `out()`. L'opération `out()` permet d'envoyer une valeur sur le canal, l'opération `in()` permet d'obtenir une valeur depuis le canal. La communication est synchrone, c'est-à-dire que `in()` bloque s'il n'y a pas de `out()` en attente, et que `out()` bloque s'il n'y a pas de `in()` en attente. La communication est 1-1 : s'il y a plusieurs `in`, un `out` donnera une valeur à un seul d'entre eux, et inversement avec plusieurs `out()` en attente et un `in()` qui arrive.

Un point original est qu'un canal est un objet transmissible, comme toute valeur : on peut envoyer un canal sur un canal (cf go.test.TestShm20).

Un canal possède en plus une opération d'observation : `observe(direction)` demande à être prévenu s'il y a une demande de `in` / `out` (selon la direction demandée) par un mécanisme de *callback*. Une observation disparaît si elle est déclenchée. Noter qu'il peut y avoir plusieurs observateurs pour un même canal.

Enfin, il est possible d'attendre qu'une action in/out soit réalisable parmi un ensemble de canaux. L'interface `go.Selector` fournit une opération `select` qui renvoie un canal sur lequel l'action écoutée est faisable, et qui bloque tant qu'aucune action n'est faisable.

2 Organisation du code

Vous allez réaliser trois implantations. Le code est donc structuré ainsi :

`go.Channel` : interface générique d'un canal.

`go.Selector` : interface d'un sélecteur.

`go.Direction` : énumération définissant les directions `In` et `Out`.

`go.Observer` : interface d'un observateur (*callback* pour `Channel.observe`.

`go.Factory` : interface pour créer un canal ou un sélecteur.

Le code utilisateur instancie un `go.Factory` de l'implantation qu'il veut utiliser (p.e. un `go.shm.Factory`) puis utilise `newChannel` ou `newSelector`. Ainsi, le code utilisateur ne référence qu'en un unique point l'implantation qu'il utilise.

3 Version en mémoire partagée

Il s'agit de réaliser une implantation qui s'exécute directement dans la même machine virtuelle que les codes utilisateurs. Il s'agit de réaliser les trois implantations `go.shm.Channel`, `go.shm.Selector` et `go.shm.Factory`.

Conseils : il est conseillé de réaliser le projet par étapes :

1. Implanter `go.shm.Channel::in` et `go.shm.Channel::out`. La synchronisation correcte demande un peu de réflexion.
2. Implanter `go.shm.Factory::newChannel` et tester abondamment.
3. Implanter `observe`.
4. Implanter `go.shm.Selector` et `go.shm.Factory::newSelector`.

Attention : les exemples fournis ne sont pas des tests suffisants, loin de là !

4 Version client / mono-serveur

Il s'agit de réaliser une implantation qui utilise un serveur qui centralise le contenu des canaux. Les clients accèdent au serveur par RMI. L'implantation du serveur utilise en interne la version des canaux en mémoire partagée.

Conseils : il est conseillé de réaliser le projet par étapes :

1. Implanter `go.cs.Channel`, `go.cs.Factory::newChannel`, le serveur (`ServerImpl`) et les autres classes nécessaires pour faire fonctionner `in`/`out`.
2. Réaliser le mécanisme d'observation.
3. Implanter `go.cs.Selector` et `go.shm.Factory.newSelector`.

Attention : le nom de la classe serveur `go.cs.ServerImpl` ne doit pas être changé, il est fixé pour les tests automatiques. Vous pouvez bien sûr ajouter de nouvelles classes ou interfaces.

5 Version décentralisée avec des sockets

Il s'agit de réaliser une implantation où chaque canal est situé sur un site, et des utilisateurs de ce canal sont situés sur d'autres sites. La communication est faite par des sockets. On ne demande que la réalisation de `in()` et `out()` et, optionnellement de `observe`. `select` n'est pas à faire.

Proposition d'architecture

- Un canal est constitué de deux implantations :
  - Un `go.sock.ChannelMaster` qui accepte des connexions (socket serveur) où il reçoit des requêtes de `in` et `out`. Il encapsule un `go.shm.Channel` pour l'implantation effective des opérations.
  - Un `go.sock.ChannelSlave` qui envoie ses requêtes de `in` et `out` au `ChannelMaster`.
- Un service de nommage (à implanter) permet aux `ChannelSlave` de trouver l'adresse (adresse IP + numéro de port) du `ChannelMaster`.
- Lors de l'appel de `Factory::newChannel`, on interroge le service de nommage et on crée un `ChannelMaster` si le canal n'existe pas ; s'il existe, on crée un `ChannelSlave`.

Note : pouvoir envoyer un canal dans un canal demande d'adapter la sérialisation.

6 Modalités pratiques

Projet réalisé *en binôme*. Les séances de suivi sont obligatoires : un point d'avancement est fait à chaque fois. Un test aura lieu le 5 juin 2024.

Squelette du code : sur moodle.

Vous devez rendre le 5 juin 2024 :

— un *petit* rapport présentant l'architecture, les algorithmes des opérations essentielles, une explication claire des points délicats et de leur résolution, et un mot sur les exemples originaux développés ;

— le code complet, y compris le plan de test et les tests que vous avez développés.

Attention : les exemples fournis (`TestShm01`, `TestShm11` en mémoire partagé, `TestCS01a` et `TestCS01b` en client-serveur) doivent compiler et s'exécuter correctement *sans que vous y touchiez le moindre caractère*. Ces tests valident votre respect de l'API. Ils ne sont cependant absolument pas exhaustifs et vous devez développer vos propres tests.

Organisation des codes sources

Package `go` : les interfaces abstraites ;

Package `go.shm` : implantation en mémoire partagée ;

Package `go.cs` : implantation distante avec un serveur centralisé ;

Package `go.sock` : implantation distribuée avec des socket ;

Package `go.test` : quelques tests fournis ;

Package `go.whiteboard` : un exemple d'application ;

Package `go.`*`autre`* : ce que vous voulez.