Systèmes concurrents & intergiciels
Projet

Philippe Quéinnec

ENSEEIHT
Département Sciences du Numérique

23 février 2024

Plan

Objectif

Réaliser une implantation de la communication par canaux synchrones (Go, π-calcul)

1. En mémoire partagée
2. En client - serveur (un serveur, plusieurs clients concurrents)
3. En décentralisé (plusieurs serveurs)

Sujet, présentation et code fourni sur moodle.

Communication par canaux synchrone

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

Exemple

```
Channel<Integer> c = ...

new Thread(() -> {
        System.out.println("Out ready");
        c.out(4);
        System.out.println("Out done");
}).start();

new Thread(() -> {
        System.out.println("In ready");
        int v = c.in();
        System.out.println("In done" + v);
}).start();
```

Exemple avec envoi de canal

```
Channel<Channel<Integer>> c1 = ...

new Thread(() -> {
        Channel<Integer> c2 = ...
        c1.out(c2);
        int v = c2.in();
}).start();

new Thread(() -> {
        Channel<Integer> c = c1.in();
        c.out(4);
}).start();
```

Observation d'un canal

```
/** A class can implement the Observer interface when it
  *  wants to be informed of changes in a channel. */
public interface Observer {
    public void update();
}
```

`Channel::observe(Direction dir, Observer obs)`

- Ajoute un observateur pour ce canal, pour la direction spécifiée
- `obs.update()` est invoqué quand une opération in/out est présente (selon la direction demandée)
- L'observation n'est déclenchée qu'une fois, puis oubliée.

Exemple d'observation

```
Channel<Integer> c = ...

class MyObservation implements Observer {
  public void update() {
    System.out.println("Un in() est là");
  }
}

new Thread(() -> {
    c.observe(Direction.In, new MyObservation());
    ...
}).start();

new Thread(() -> {
    int v = c.in();
}).start();
```

Factory

Une implantation de l'interface `Factory` permet de créer un canal :

```
package go;
public interface Factory {
    /** Création ou accès à un canal existant. */
    public <T> go.Channel<T> newChannel(String name);
}
```

```
Factory factory = new go.shm.Factory();
Channel<Integer> chan1 = factory.newChannel("c1");
Channel<Integer> chan2 = factory.newChannel("c2");
```

En changeant de Factory, on peut changer l'implantation des canaux utilisés.

Sélection parmi un ensemble

- `select` : attente qu'une opération soit possible parmi un ensemble de in/out sur des canaux.
- `select` renvoie un canal (parmi ceux spécifiés) où l'action demandée est faisable sans bloquer.
- si plusieurs réponses sont possibles, l'une arbitraire est fournie.

```
public interface Selector {
    public Channel select();
}
```

```
public interface Factory {
  /** Canaux écoutés et la direction pour chacun. */
  Selector newSelector(Map<Channel, Direction> channels);
  /** Canaux écoutés et la même direction pour tous. */
  Selector newSelector(Set<Channel> channels, Direction dir);
}
```

Sélection : exemple

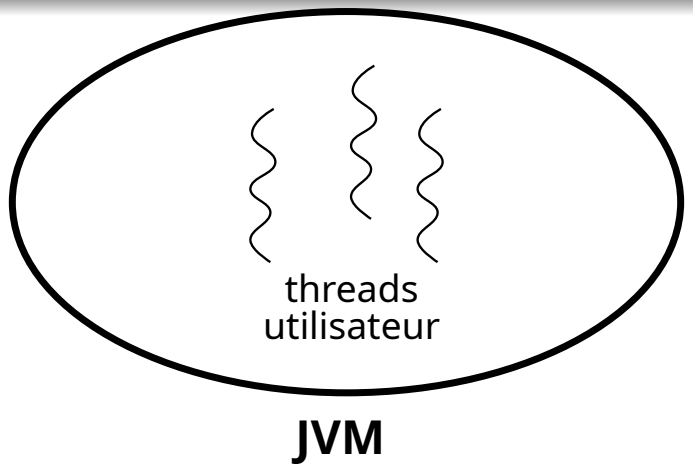
```
Factory factory = new go.shm.Factory();
Channel<Integer> c1 = factory.newChannel("c1");
Channel<Integer> c2 = factory.newChannel("c2");
Channel<Integer> c3 = factory.newChannel("c3");
Selector s = factory.newSelector(java.util.Set.of(c1, c2, c3),
                                 Direction.In);

new Thread(() -> { c1.out(4); ... c2.out(6); }).start();

new Thread(() -> {
    Channel<Integer> c = s.select();  // will get c1
    int v = c.in();
    ...
    Channel<Integer> cc = s.select(); // will get c2
    v = cc.in();
}).start();
```

Plan

Réalisations

1. Version en mémoire partagée :
   Noyau et plan de tests
   (les tests fournis sont des exemples et sont très insuffisants)
2. Version client - serveur
3. Version décentralisée

Version en mémoire partagée

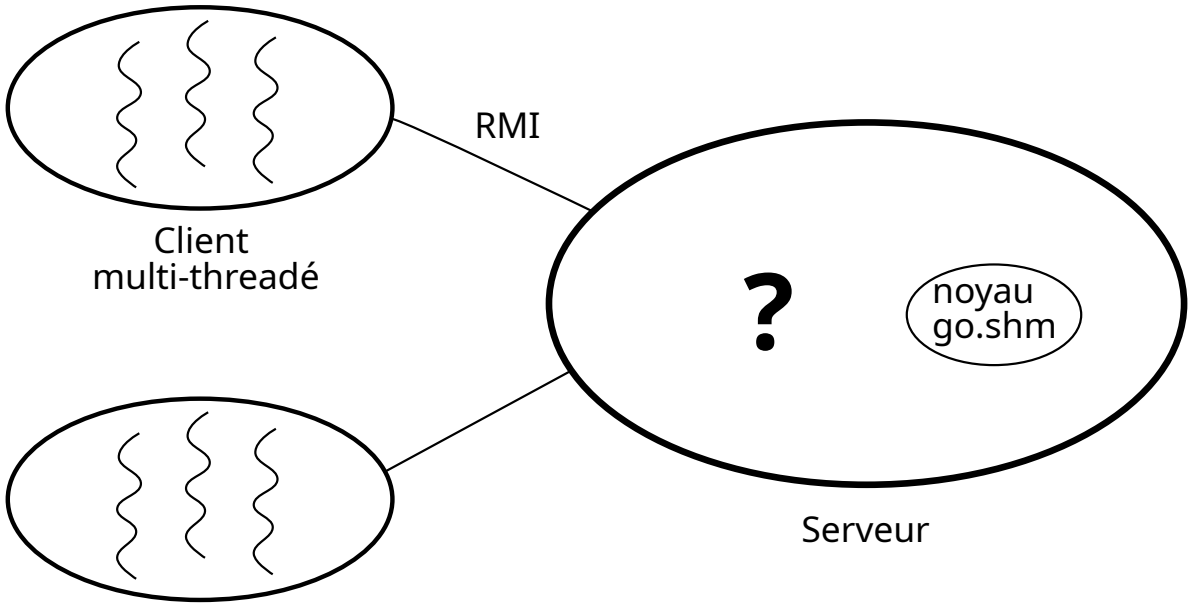


Plusieurs activités (threads), dans la même machine virtuelle Java.

1. Implanter `go.shm.Channel::in` et `go.shm.Channel::out`.
2. Implanter `go.shm.Factory::newChannel` et tester abondamment.
3. Implanter `observe`.
4. Implanter `go.shm.Selector` et `go.shm.Factory::newSelector`.

Version client - serveur

Un serveur stockant le contenu des canaux, un ou des clients distants accédant au serveur. Chaque client peut lui-même être concurrent (multi-threadé).



- Réutiliser dans le serveur l'implantation des canaux en mémoire partagée
- La difficulté est dans les observations

Version décentralisée

- Un canal est constitué de deux implantations :
  - Une partie maître, qui encapsule un `go.shm.Channel`, et accepte des requêtes de `in` et `out` ;
  - Une partie esclave, qui envoie ses requêtes de `in` et `out` au maître.
- La communication est réalisée par socket.
- Un service de nommage permet de savoir si un maître existe et où il est.
- `Factory::newChannel` crée un maître si le service de nommage ne connaît pas le nom, un esclave sinon.

Note : l'observation de canal est optionnelle et `select` n'est pas demandé (trop complexe).

Plan

Modalités

- Projet exécuté *en binôme*
- Date limite de constitution des binômes : pas d'urgence
  Envoyés à `queinnec@enseeiht.fr`
- Séances de suivi et d'assistance, présence obligatoire
- Service de tests automatiques pendant les séances
- Rendu le 5 juin 2024 sur moodle
- Séance de test le 5 juin 2024

Modalités (2)

- Squelettes et code de base disponible sur moodle
- Respectez l'API : interfaces `Channel`, `Factory`, `Selector`, etc ;

Les tests fournis doivent compiler et s'exécuter correctement sans que vous y touchiez le moindre caractère.