RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

TOULOUSE INP N7

Projet intergicielle

Rapport - Février 2025

Killian Falguiere

www.enseeiht.fr

Table des matières

Table des figures

Table des listings

Introduction

Ce projet consiste à implémenter un équivalent des Canaux du langage Go en Java. Ce programme sera fait en trois versions : une version locale et centralisée en mémoire partagée, une version accessible à distance en mode Client/Serveur à l'aide de RMI en se basant sur la version précédente, et une dernière version décentralisée utilisant les sockets.

Un canal permet d'échanger des valeurs/références d'objets d'un type fixé à la création du canal. La lecture et l'écriture sont synchronisées. Un canal offre trois fonctions :

- Une fonction d'écriture « out ».
- Une fonction de lecture « in » d'une valeur écrite dans le canal.
- Une fonction permettant de d'inscrire un observeur associé à une action précise (in/out) à un canal. L'observeur est mis-à-jour quand l'action observé est appelé.

Nous aurons également un sélecteur qui nous permet d'inscrire une ensemble de canaux et de directions que l'on veut observer. Une fonction « select » nous permet de récupérer un canal dont une opération de la direction opposée à celle initialement donnée a été observée. Plus simplement, un sélecteur nous permet de récupérer un canal dont l'opération que l'on souhaite faire dessus ne sera pas bloquée.

Mémoire Partagée

Architecture générale

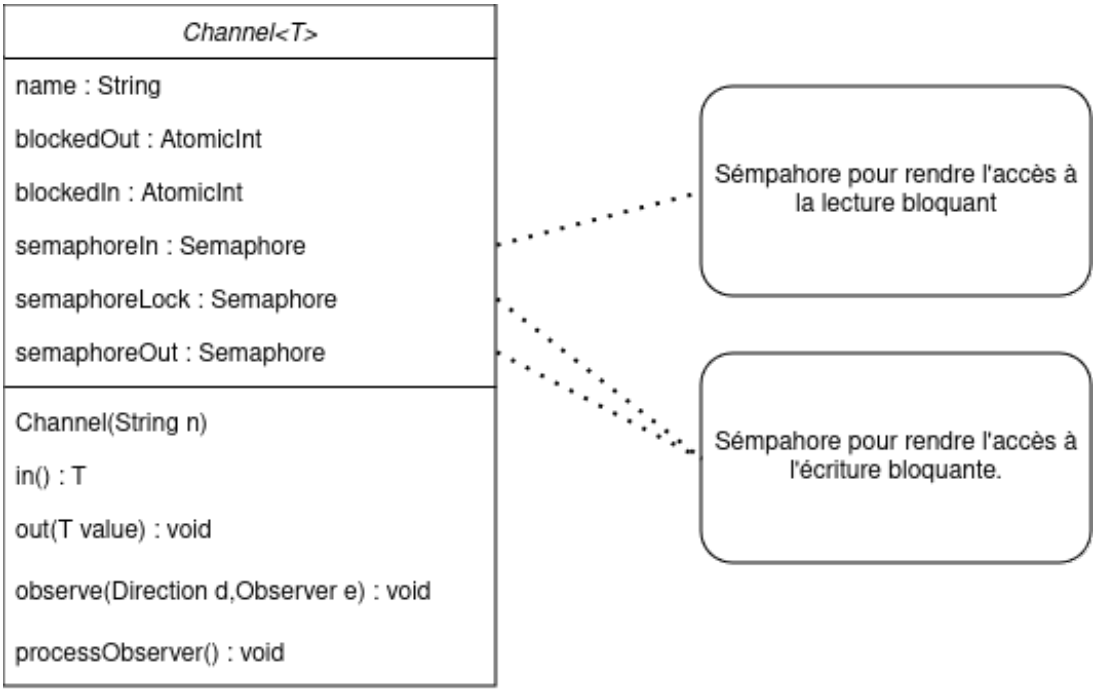

Figure 1. – Représentation du Channel Shared Memory

Nous avons choisies de rendre les actions bloquantes à l'aide de sémaphore. Leur avantage étant d'être simple à utiliser et que leurs actions sont lisibles (on sait rapidement quelle zone bloquée est libérée). L'inconvénient principal étant qu'ils doivent être instanciés avec une valeur initiale qui

permet de déterminer s'il sera possible d'acquérir un sémaphore initialement. Ce principe de valeur du sémaphore peut provoquer une multiplication des sémaphores à utiliser pour compenser certains cas.

Par exemple, 2 processus devant se synchroniser pour atteindre une même section relativement au même moment:

```
Semaphore semaphore1 = new Semaphore(1);
Semaphore semaphore2 = new Semaphore(1);

Séquencement   | Proc1                  | Proc2
1              | semaphore1.acquire()   |
2              | /* section conjointe */| semaphore2.acquire()
3              | semaphore2.release()   | // section conjointe
4              |                        | semaphore1.release()
```

Listing 1. – Démo du fonctionnement des sémaphores

En mettant les deux sémaphores à 1 initialement on retire tout aspect bloquant. En les mettant à 0 le programme sera bloqué sans pouvoir rien faire. On se retrouve donc à devoir mettre un sémaphore à 1, l'autre à 0 et à ajouter un nouveau sémaphore qui servira à bloquer le processus avec la sémaphore à 1 APRES qu'il ait débloqué l'autre processus.

```
Semaphore semaphore1 = new Semaphore(1);
Semaphore semaphore1_2 = new Semaphore(0);
Semaphore semaphore2 = new Semaphore(0);

Séquencement   | Proc1                           | Proc2
1              | semaphore1.acquire()            |
2              |                                 | semaphore2.acquire() // bloqué
3              | semaphore2.release()            | // débloqué
4              | // section conjointe            | // section conjointe
5              | semaphore1_2.acquire() //bloqué |
6              | /* débloqué */                  | semaphore1_2.release()
7              | /* boucle suivante */           | semaphore1.release()
```

Listing 2. – Démo de l'utilisation des sémaphore dans un channel

C'est exactement cette méthode que nous avons utilisée pour les fonctions in/out respectivement Proc2 et Proc1. Le méthode select du Selector est également protéger à l'aide d'un sémaphore.

Observer

Les AtomicInt permettent à la fonction observe de vérifier si une action est bloqué. Par exemple, on incrémente blockedOut juste avant le acquire() du premier sémaphore et on le décrémente juste après le acquire() du deuxième sémaphore. Dans la fonction « observe », un observeur qui s'inscrit doit être immédiatement mis-à-jour si l'action surveillée était déjà bloqué. On considère qu'une action est bloqué si et seulement si l'AtomicInt lui correspondant est supérieur à 0 et que l'AtomicInt opposé est à 0. C-à-d, le « out » est considéré bloqué si blockedOut > 0 et blockedIn == 0 et

inversement. Ceci sert à bloquer les cas d'une mis-à-jour instantanée d'un observeur qui aurait été inscrit lors de la mise-à-jour d'un autre observeur :

```
Channel canal1 = new Channel("c1");
canal1.observe(Direction.In, () -> {
      canal1.observe(Direction.In, () -> { print("Ok") })
}) // Cet oberveur n'est pas immédiatement appelé car aucune action bloquée

new Thread() {
    canal1.out(1); //Incrémente blockedOut
}.start();
Thread.sleep(200);
canal1.in();
/* Ceci va
- Incrémenter blockedIn
- MAJ l'observeur1
      - L'observeur1 inscrit un nouvel observeur sur In
      - Si on considère In bloqué seulement quand blockedIn > 0 alors on MAJ
l'observeur
      - En revanche si on considère In bloqué quand blockedIn > 0 ET blockedOut
== 0 alors l'observeur ne sera pas MAJ. */
```

Listing 3. – Code d'un cas spécial avec les observeurs

Nous avons choisis d'utiliser les deux AtomicInt dans la vérification du *pending* puisque nous considérons que l'action n'est pas réellement « bloquée » à ce moment mais en cours. Il s'agit donc d'une interprétation de la spécification.« «

Selecteur

Notre sélecteur fonctionne comme une pile de canaux avec des actions disponibles. Pour cela nous avons choisis d'utiliser une liste chaînée. La fonction LinkedList::addFirst permet d'ajouter un élément en tête de liste tandis que LinkedList::poll retire et retourne l'élément en tête de liste, de manière assez proche de Pop et Push d'une pile (même si nous utilisons getFirst() au lieu de poll et que la suppression est faite ultérieusement. Notre select() agit donc comme une LIFO/FILO.

Le sélecteur se base sur 2 observeurs :

- Un observeur sur la direction opposée à celle demandée à l'inscription, qui ajoute le canal dans la liste chaînée lors que l'action dans la direction demandée est disponible.
- Un observeur sur la direction demandée qui retire le canal de la liste et ré-inscrit l'observeur de la direction opposée après (étant donné que le canal purge les observeurs après la MAJ).

Plan de tests

Notre plan de test est principalement composé de tests d'intégration puisque le comportement individuel des fonctions est trivial et/ou peu intéressant (et difficilement testable puisque souvent bloquante), comparé au fonctionnement globale du paquet **Shm** et de ses classes.

Tous les tests ont un préfixe à leur nom « **testShm** ».

Nom du test	Objectif
Custom01	Vérifie qu'un même canal peut-être retourné plusieurs fois par le même sélecteur si l'action out qui nous intéresse est disponible plusieurs fois
Custom02	Vérifie que l'observe est bien update directement quand l'action observé est pending
Custom03	Teste un cas particulier d'entrelacement avec deux observe : l'update dans les observes doit être effectué avant l'action qui appelle l'update
Custom04	Comme pour Custom1 mais avec l'action in
Custom05	On vérifie que le select fonctionne bien en FILO
Custom06	Test simple de Channel dans Channel
Custom07	Test du sélecteur avec Channel différent (Channel, Integer, Booleen)
Custom08	Vérifie que le sélecteur retourne bien les bonnes instances
CustomBourrin01	Créé 10 Channel qui vont in indéfiniment, puis vérifie que l'on deadlock pas si on out dans chaque Channel choisis aléatoirement en boucle pendant 3 seconde
CustomBourrin02	Comme CustomBourrin01 mais on inverse les actions
CustomBourrin03	Test intense d'in et out sur un même Channel : on vérifie la cohérence de ce qu'on lit par rapport à ce qu'on écrit

Client/Serveur (RMI)

Architecture générale

Pour utiliser RMI, il va être nécessaire de faire soit appel à des classes sérialisables soit à des classes accessibles par référence à distance (Interface Remote, hérite d'UnicastRemoteObject). Cependant nous ne pouvons pas modifier les interfaces qui nous ont été fournis. Notre choix s'est donc porter sur une architecture basée sur la délégation (appelé proxies dans le code).

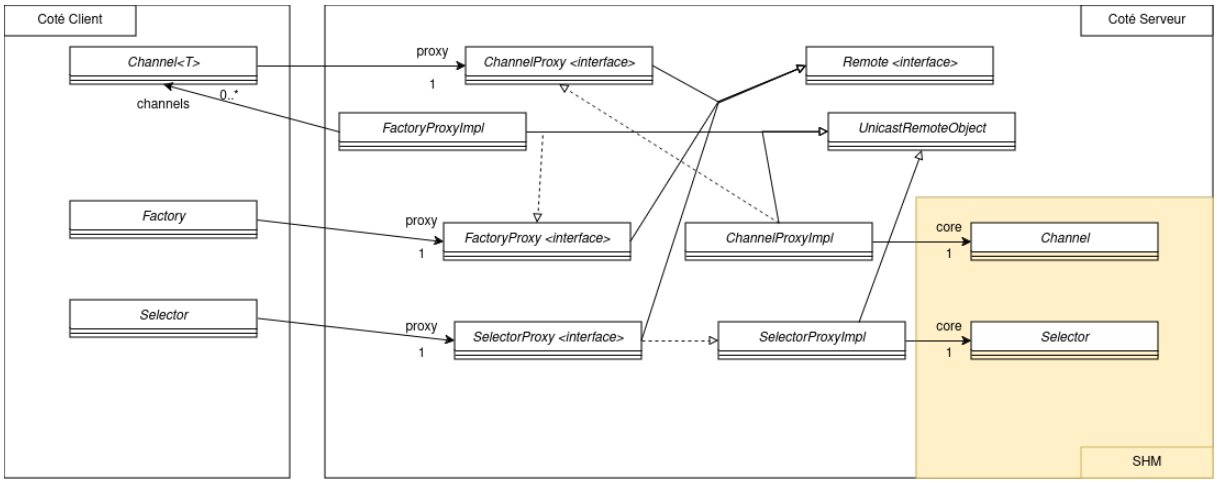

Figure 2. – Architecture du CS

Les classes Channel, Factory, Selector (go.cs) implémentent les interfaces fournis et seront accessibles sur le client. Ces classes ont toutes une référence vers un interface proxy/délégué via RMI qui est situé sur le serveur. La FactoryProxyImpl retourne des Channel et des Selector (go.cs) qui sont donc Serializable. SelectorProxy::select() retourne également un go.cs.Channel. Le SelectorProxyImpl et ChannelProxyImpl ont une référence vers leurs équivalents dans SHM et qui leur sert de moteur.

Observer

L'Observer fonctionne sur un principe similaire mais inversé : c'est le client qui propose un service d'appel de fonction à distance au serveur en lui fournissant une référence d'un objet accessible via RMI avec une fonction *update* (un ObserverProxy qui encapsule un java.util.Observer). Le serveur va ensuite utiliser cette référence dans une instance de java.util.Observer qui sera passé en argument du shm.Channel.*observe*.

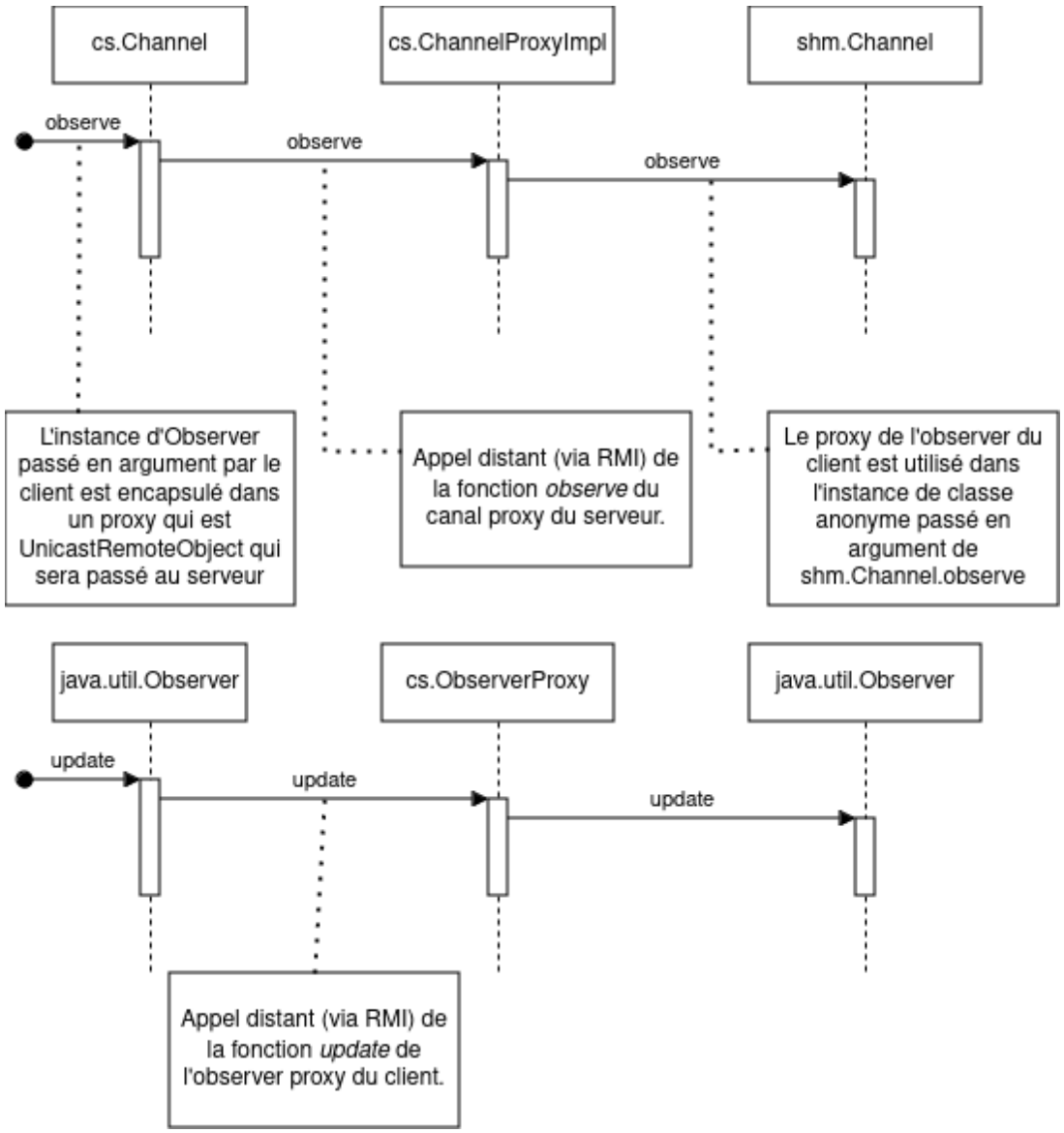

Figure 3. – Séquence d'observe et update

Plan de tests

De la même manière que nos tests *Shm*, notre plan de test est principalement composé de tests d'intégration.

Tous les tests ont un préfixe à leur nom « **testCS** ».

Nom du test	Objectif
Custom01	Vérifie qu'un observer en direction in fonctionne correctement
Custom02	Vérifie qu'un observer en direction out fonctionne correctement
Custom03	Vérifie qu'un observer enregistré après une action in est appelé
Custom04	Vérifie qu'un observer enregistré après une action out est appelé

Nom du test	Objectif
Custom05	Vérifie que l'enchaînement de in/out sur un canal fonctionne correctement
Custom06	Vérifie que l'enchaînement de in/out avec 2 observer(1 sur in et l'autre sur out) sur un canal fonctionne correctement
Custom07	Vérifie que l'enchaînement de in/out avec 1 observer sur un canal fonctionne correctement
Custom08	Vérifie que le selector fonctionne avec 1 canal et 2 sélections
Custom09	Vérifie que le selector fonctionne correctement avec 3 canaux
Custom010	Vérifie que l'enchaînement de in/out sur 3 canaux fonctionne correctement
Custom011	Vérifie que les selector retournent bien une instance identique aux canaux d'origine

Canaux décentralisés (Socket)

Architecture

Pour la version socket, nous étions libre sur les choix de conception. Nous avons alors fait le choix de partir sur une version décentralisés. Ce sont aux clients d'héberger les canaux et non un serveur central. Dans notre cas, le premier client à demander la création d'un canal x sera celui qui l'hébergera. Le serveur de Naming (équivalent à un serveur DNS) gardera en mémoire les informations de ce client pour les renvoyer à d'autres clients si besoin comme le montre la Figure 4.

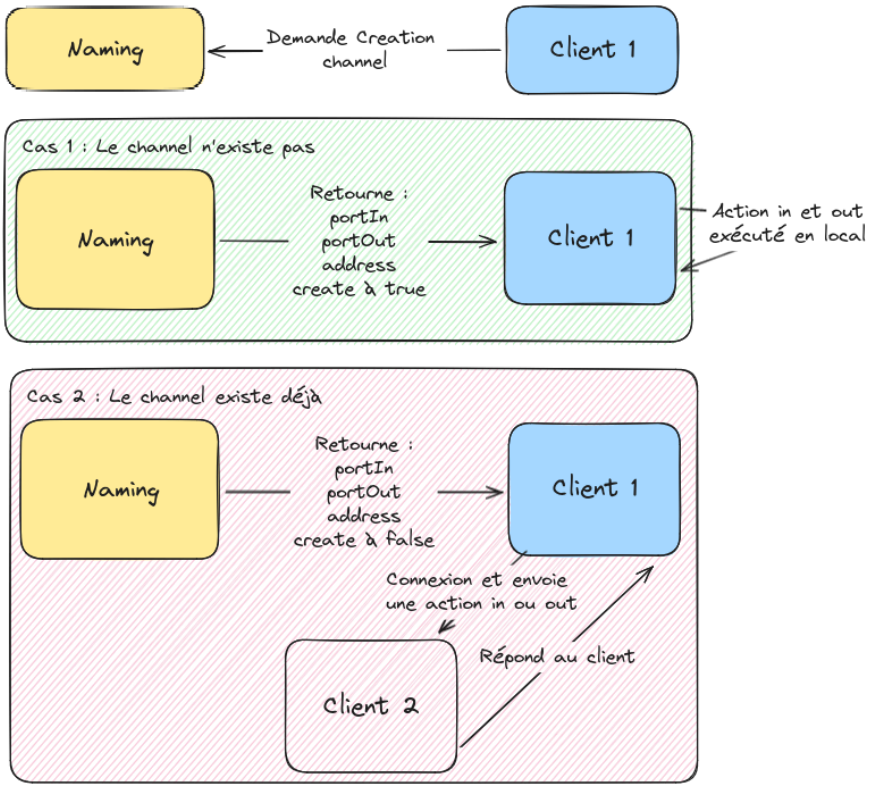

Figure 4. – Architecture de la version Socket

Pour gérer les actions *in / out*, nous avons fait le choix d'utiliser un serveur socket par action. Donc, le serveur de Naming et les clients identifieront un canal de la manière suivante :

- un nom
- un port pour l'action *in*
- un port pour l'action *out*
- une adresse distante

Par la suite, si un client souhaite utiliser l'action d'un canal, il se connectera au port lié à cette action et discutera avec le client « maître », comme le montre la Figure 5.

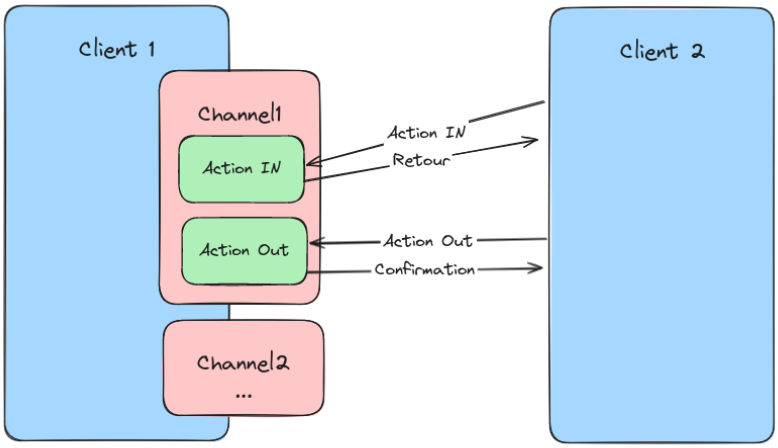

Figure 5. – Discussion entre 2 clients

Chaque action à son port, donc son serveur socket. Par conséquent, un canal a deux serveur socket assignés. Cela permet de bloquer le client si le canal est bloqué et de mettre en place une FIFO sur les actions.

Ce choix a été totalement arbitraire, nous aurions pu utiliser un serveur socket unique pour gérer les deux actions et lancer des Thread pour chaque connexion. Cependant, d'après nous, cette solution était moins stable et maintenable.

Plan de tests

De la même manière que nos tests `Shm` et `RMi`, notre plan de test est principalement composé de tests d'intégration.

Tous les tests ont un préfixe à leur nom « **testSocket** ».

Nom du test	Objectif
Custom01	Vérifie que l'action in fonctionne correctement
Custom02	Vérifie que l'action out fonctionne correctement
Custom03	Vérifie que l'enchaînement de in / out fonctionne correctement et est bien ordonné
Custom04	Vérifie l'enchaînement de in / out fonctionne correctement à travers trois canaux et trois processus

Livrables

- **sources.zip** - Fichier contenant le code source du projet