Laetitia PHAM
Maxence BRUNET

Enseignant : *Jacques* MALENFANT
Groupe : LAMA

UE CPS

Compte-Rendu

Juin 2020

Science et Technologie du Logiciel
Sorbonne Université
Année 2019/2020

1 Composants, Ports et Interfaces

1.1 Les composants utilisées

Le composant *Broker*, situé dans le répertoire */src/components/Broker.java*, transite les messages vers les souscripteurs et éventuellement vers d'autres courtiers dans le cas d'une multi-jvm. De plus, pour une gestion avancée de la concurrence, la gestion des messages et la gestion des souscriptions sont confinées dans des classes java (respectivement */src/annexes/TopicKeeper.java* et */src/annexes/Gestion-Client.java*). Leur sûreté est assurée par le principe des caches de données et l'utilisation de collections synchronisées.

Le composant *Publisher*, situé dans le répertoire */src/components/Publisher.java*, publie des messages dans différents sujets de différentes machines virtuelles.

Le composant *Subscriber*, situé dans le répertoire */src/components/Subscriber.java*, reçoit les messages (des courtiers) des sujets auquel ils sont abonnés.

1.2 Interfaces

La transmission des messages d'une copie de courtier à l'autre est une opération qui relève de la gestion interne des copies de courtiers. Nous avons choisi de définir une nouvelle interfaces pour permettre cette gestion. Deux nouvelles interfaces apparaissent dans notre répertoire *src/interfaces* (paquetage interfaces):

- *TransfertImplementationI* : Étend la classe java *java.rmi.Remote*, lui permettant d'appeler des méthodes distantes (via des machines virtuelles).
- *TransfertCI* : Étend les interfaces OfferedI, RequiredI et TransfertImplementationI. Cette interface est offerte et requise, car elle est utilisée par les courtiers qui non seulement reçoivent des informations, plus précisément des messages sérialisables, mais aussi envoie des informations vers d'autres courtiers.

1.3 Ports

Pour transiter un message d'un courtier sur une machine A vers un autre courtier sur une machine B, on crée deux nouveaux ports (situé dans le répertoire */src/ports*) :
- *TransfertCInBoundPort* : étend notre interface TransfertCI qui a été expliqué précédemment.
- *TransfertCOutBoundPort*: étend également notre interface TransfertCI.

2 Greffons

Nous avons essayé d'implémenter des greffons dynamique en s'inspirant de *dconnection.example* dans *examples-basis-plugins*.

```
public class BrokerPublicationDynamicPlugin
extends DynamicConnectionServerSidePlugin{
        @Override
        protected InboundPortI createAndPublishServerSideDynamicPort
                                (Class<?> offeredInterface) throws Exception {
                assert PublicationCI.class.isAssignableFrom(offeredInterface) &&
                offeredInterface.isAssignableFrom(PublicationCI.class);
                InboundPortI pibp = new PublicationCInBoundPort(this.owner);
                pibp.publishPort();
                return pibp;
        }
}
```

Cependant, en utilisant cette structure nous aurions perdu l'utilisation des pools de threads créer dans le *Broker* qui était indiqué dans la création de *PublicationCInBoundPort* comme deuxième argument.

Ainsi, nous avons décidé d'utiliser les greffons déjà existant des étapes précédentes. Ceux-ci étant situés dans le répertoire *src/plugins*.

3 CVM Multi-Clients

Nous avons mis en place un nouveau launcher *"CVMTestMultiClients.java"* présent dans le répertoire *src/launcher* (paquetage launcher).

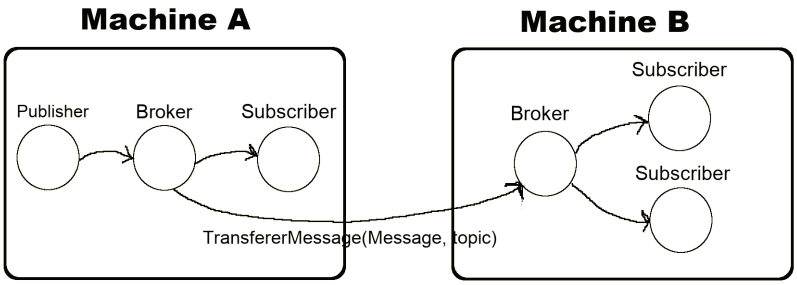
Cette CVM a pour but de créer plusieurs clients et de vérifier que les messages publiés par plusieurs producteurs soient bien reçus par tous les souscripteurs abonnées à ce sujet. Cela vérifie également que les souscripteurs ne reçoivent pas plusieurs fois le même message. De plus, les tests liés à cette CVM reste les mêmes que ceux de la première et deuxième étape du cahier des charges.

4 DCVM et Multi-JVM

Notre classe DCVM se situe aussi dans le répertoire *src/launcher*. Pour mettre en place cela, un deuxième constructeur a été ajouté dans le composant *Broker*. En effet, celui-ci permet d'initialiser les ports de transferts pour la communication entre les courtiers, contrairement au premier constructeur.

Par la suite, nous avons créé plusieurs machines virtuelles composées elles-mêmes d'un courtier, de producteurs et de consommateurs. L'objectif est de faire transiter des messages (sérialisables) d'une machine virtuelle à une autre en passant par les courtiers.

Après avoir créé les différents composants de chaque machine virtuelle, nous effectuons une interconnexion entre les courtiers via les ports de transferts (cf 1.3 Ports). Lorsqu'un producteur d'une machine A publie un message dans un sujet, le courtier traite le message et l'envoie à ses souscripteurs qui ont souscrit à ce sujet. Puis le message est transité vers un autre courtier d'une machine B en appelant la méthode *transfererMessage* via le port *TransfertOutboundPort*. Si le courtier de la machine B a déjà reçu le message, celui-ci ne fait rien sinon il le publie et l'envoie au courtier d'une autre machine continuant ainsi le cycle de transfert.



5 Test d'intégrations pour la DCVM

Nous avons effectué de nombreux scénarios afin de couvrir au maximum les différentes fonctionnalités proposées entre les composants des différentes machines virtuelles. Pour lancer ces tests et avoir un rendu visuelle, il suffit de lancer les scripts suivant sur un terminal chacune: *./start-cyclic-barriere*, *./GlobalRegistery*, puis *./start-dcvm [nom de la machine virtuelle (par exemple jvm_1)]*.

Les scénarios liés au *Publisher* sont les suivant:

- Scénario 1 : Le producteur de la machine A publie deux messages dans les sujets Peche&Cuisine et Nature&Decouverte. Le producteur de la machine B publie deux messages dans le sujet Automobile. Ce

scénario est lié avec celui du scénario 1 du *Subscriber*. Il permet de vérifier que tous les souscripteurs (de la machine A et B) abonnées à ces sujets, reçoivent les messages émis par chacun des producteurs (de la machine A et B).

- Scénario 2 : Le producteur de la machine A publie un message avec une propriété "Thon" dans le sujet "Peche&Cuisine". Le producteur de la machine B publie 100 messages dans le sujet "Automobile". Ce scénario est lié avec celui du *Subscriber*. Il permet de vérifier que ces nombreux messages soient bien reçus par les souscripteurs ayant souscrit au sujet "Automobile" et ce, même s'ils se trouvent sur différentes machines.

Par défaut, d'autres scénarios sont mis en place pour tester leurs fonctionnements. Pour tester ou non ces scénarios, il suffit de retirer ou ajouter leur numéro de scénario dans la liste des scénarios testés que ce soit dans *Publisher* ou *Subscriber*. Il ne faudra pas oublier d'exporter le projet en .jar du projet complet et le mettre dans le répertoire *src/launcher/jars* pour voir les changements.

6 Test de performances

L'objectif de cette étape étant de mesurer le nombre de publications par seconde qui sont reçues et stockées par le courtier, puis le nombre de livraison de ces messages par seconde aux abonnées devant les recevoir, nous avons réalisé plusieurs tests.

6.1 Tests:

Les tests réalisés concernent un *Publisher* qui publie 100 messages dans le sujet Automobile, deux *Broker* (un dans chaque JVM) et un total de quatre *Subscribers* (un dans la première jvm et trois dans la seconde). Un autre test a été réalisé avec l'utilisation de 7 *Publishers*, 2 *Broker* et 7 *Subscribers* et amènent à un résultat similaire.

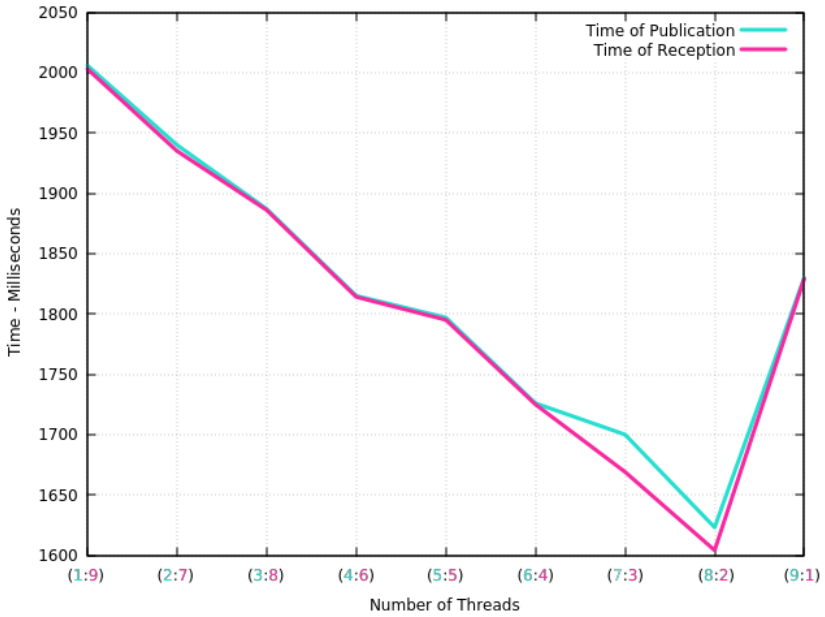


Figure 1: Les temps de publications et réceptions de 100 messages.

Pour obtenir le temps de publication pour 100 messages, nous avons fait la différence entre le moment (le

temps *epoch unix*) où le premier message est envoyé et le moment où le dernier message est reçu et stocké. En ce qui concerne le temps de réception (ou d'envoi) de ces 100 messages, nous avons réalisé un calcul entre le moment où le premier message est envoyé aux abonnées et le dernier messages reçu pas les abonnées.

Par ailleurs, nous avons effectué ces mesures en modifiant le "*Number of Threads*" à chaque fois. En effet, la première valeur, c'est-à-dire " *(1:9)* ", correspond au fait est que nous avons attribué qu'un seul fil d'exécution pour la publication et neuf autres pour l'envoi des messages.

6.2 Résultats:

Comme on peut le voir sur le graphique, plus le nombre de threads de publication est important et plus le temps d'envoi et de réception des messages diminue. Lorsque le nombre de threads de publications augmente, la publication des messages devient beaucoup plus rapide car ce sont plusieurs fils d'exécutions qui s'occupent des différents messages envoyés vers le *Broker*.

On remarque que le temps de la réception d'un message pour un *Subscriber* diminue également à la même vitesse. On peut expliquer ce phénomène par le fait est que pour qu'un *Subscriber* reçoit le dernier message, il faut que le *Publisher* envoie son dernier message. Le temps entre l'envoie du *Publisher* vers le *Broker* et du *Broker* vers le *Subscriber* étant presque équivalent. Cela explique les résultats similaires sur la courbe pour la publication et la réception.

Par manque de temps, nous n'avons pas mis en place de tampon. L'utilisation du tampon aurait sûrement permis de creuser l'écart entre le temps de publication vers le *Broker* et celui de réception du *Broker* vers le *Subscriber*. La méthode est de créer une fonction "EnvoieTampon()": Lorsque le nombre de messages envoyé est égal ou dépasse celui du tampon, alors on envoie les messages vers les *Subscriber* et vers les *Broker* situés dans d'autres JVM. Dans le cas où le tampon ne serait pas plein, avant la fin du temps imparti, on envoie le reste du tampon (non plein).