Université Pierre Et Marie Curie
Spécialité Sciences et Technologie du Logiciel

PROJET ALASCA - ÉTAPE 1

Data Center Manual

Auteurs :

Hacene KASDI

Marc REN

Responsable de l'UE :

Jacques Malenfant

2017-2018

Contents

List of Figures

1 Introduction

1.1 Etape 1

À titre de suggestion, la réalisation de cette étape peut elle-même se décomposer en trois temps :

1. Dans un premier temps, nous allons simuler l'exécution des applications en programmant manuellement les fonctionnalités correspondantes aux répartiteurs de requêtes ; en exécution mono-machine virtuelle Java et de créer, interconnecter et lancer tous les composants statiquement par la classe de déploiement (similaire à la classe CVM dans les exemples fournis avec le modèle de composants).

2. Dans un second temps, toujours en exécution mono-machine virtuelle Java, nous allons intégrer les fonctionnalités du contrôleur d'admission, ce qui vous demandera de ne plus créer les machines virtuelles et les répartiteurs de requêtes dynamiquement, dans le processus de traitement des demandes d'exécution d'applications au contrôleur d'admission.

3. Dans un troisième temps, nous allons à la génération dynamique du code des répartiteurs de requêtes et des machines virtuelles pour tenir compte de l'interface de soumission donnée lors de la demande d'admission de l'application.

4. Enfin, dans un quatrième temps, on va passer à une version pluri-machine virtuelle Java, en utilisant à la fois les fonctionnalités de répartition des composants sur plusieurs machines virtuelles avec RMI et celles de création et connexion dynamique de composants entre machines virtuelles fournies par le modèle à composant.

2 RequestDispatcher

Le **RequestDispatcher** est le composant qui nous permet de simuler le comportement d'un répartiteur de requêtes. Ce composant peut recevoir des requêtes et les distribue aux différentes machines virtuelles **ApplicationVM** qui lui sont reliées.

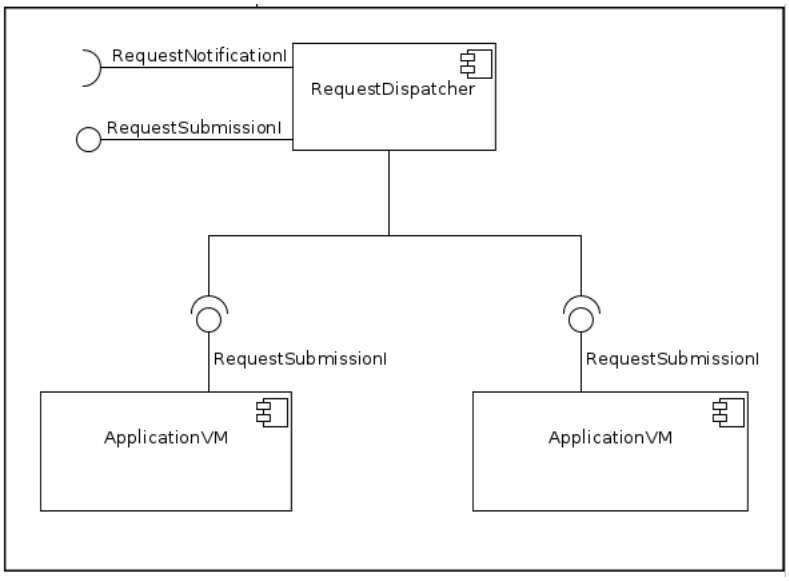

Figure 1: Request Dispatcher Component.

2.1 Les caractéristiques d'un RequestDispatcher

Ce composant est crée une fois la requête d'admission est acceptée par le **AdmissionController**. Ce composant se charge de distribuer les requêtes sur les deux **ApplicationVM** sous une politique Round Robin pour distribuer équitablement les requêtes sur les machines virtuelles, sachant que ces dernières ont allouées des ressources d'un seul computer. Chaque Machine virtuelle prend deux **Cores** et occupe un **Processor** pour exécuter ses requêtes.

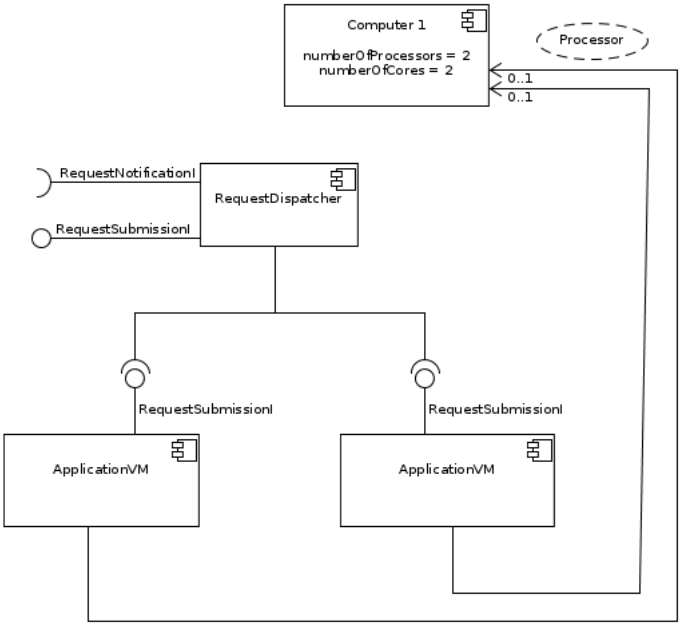

Figure 2: Request Dispatcher Component and AVM.

3 AdmissionController

Un centre de calcul offre le service d'hébergement d'application via un composant contrôleur d'admission responsable de recevoir les demandes d'exécution d'applications. Le composant a le choix d'accepter ou de refuser selon la disponibilité des ressources sur les différents Computers disponible sur le centre de calcul.

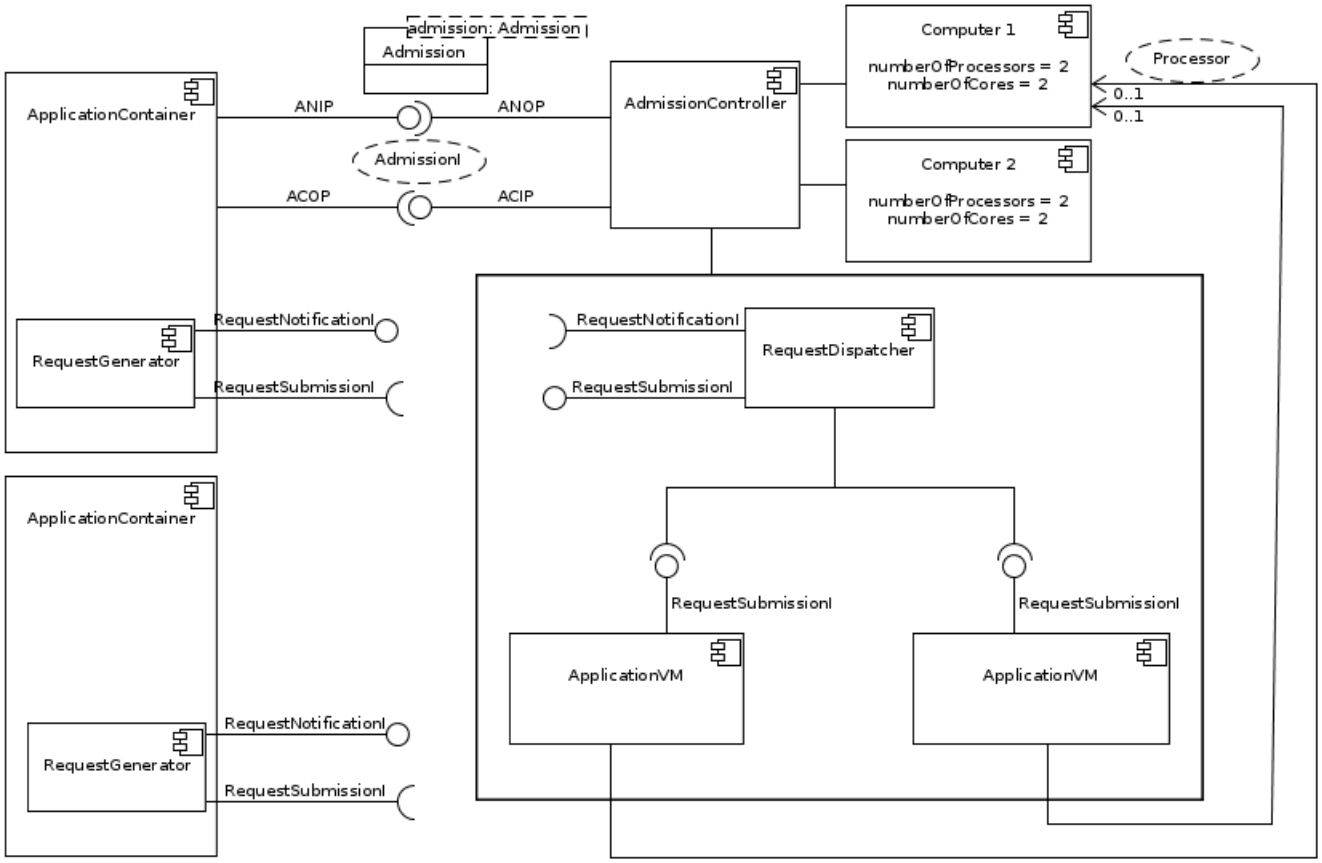

Figure 3: Admission Controller Component.

3.1 La politique de réponse aux demandes de Hosting

Le composant AdmissionController il a une connaissance sur tous les Computers disponibles sur le centre de calcul.
Il peut consulter l'état de chaque computer, pour faciliter la localisation des Computers disponibles nous avons mis en place un système pour consulter les ressources et récupère l'index du computer qui possède des Processors et des Cores non alloués

Une fois un Computer est localisé qui est susceptible de l'exécuter et initie l'exécution sur ce dernier en créant une machine virtuelle, en l'initialisant et en lançant l'exécution de l'application sur cette dernière.

Le contrôleur d'admission propose donc une interface de soumission des applications **AdmissionRequestI**, et maintient des informations sur l'utilisation courante des ordinateurs du centre lui permettant de faire cette allocation.
Il se charge de répondre aux **ApplicationContainer** soit positivement ou négativement

3.2 Contrôleur de ressources disponibles

Comme nous l'avons mentionné dans la section ci-dessus, un **AdmissionController** il a une connaissance sur l'état de tous les ordinateurs disponibles sur le centre de calcul, il consulte ces ressources lors d'une demande de Hosting qui parvienne d'une **ApplicationContainer**.
Dans le cas où des ressources sont disponible, le **AdmissionController** mis à jour la variable isAllowed de la requête **AdmissionI** et il le mit **isAllowed = true** et renvoie la requête sur le port **AdmissionNotificationOutboundPort** en invoquant la méthode **notifyAdmissionNotification(admission)** Dans le cas où des ressources sont disponible, le **AdmissionController** mis à jour la variable isAllowed de la requête **AdmissionI** et il le mit **isAllowed = true** et renvoie la requête sur le port **AdmissionNotificationOutboundPort** en invoquant la méthode **notifyAdmissionNotification(admission)**

3.3 Génération dynamique du code

Nous avons utilisé la réflexion java pour générer dynamiquement le code, également nous l'avons utilisé pour générer le code des Connecteurs.

3.4 génération du code du Connector

Nous avons utilisé le javassist pour générer dynamiquement le code java du Connector.

```
/*
 * Generate a Class Connector (AdmissionNotificationConnector) using the abstract method of the class
 * JavassistUtility using Javassist
 */
HashMap<String, String> mapMethods = new HashMap<String, String>();
mapMethods.put("notifyAdmissionNotification", "allowOrRefuseAdmissionNotification");
Class<?> admissionConnector = JavassistUtility.makeConnectorClassJavassist(
        "fr.upmc.datacenter.software.applicationcontainer.connectors.AdmissionNotifConnector",
        AbstractConnector.class,
        AdmissionNotificationI.class,
        AdmissionNotificationI.class,
        mapMethods);
```

Figure 4: Admission Notification Connector.

4 Multi JVM

4.1 Fonctionnement

Deux fichiers sont primordiaux pour faire fonctionner exécuter plusieurs **Distributed CVM**, un fichier xml de configuration et un fichier policy. Le fichier policy permet de définir les différents ports utilisés, les droits d'accès aux différents fichiers et autres. Le fichier xml de configuration permet de définir le hostname et le port de la **Barrière Cyclique**, le **serveur global RMI** et des différents **Distributed CVM**.
La **Barrière Cyclique** permet de garantir la synchronisation du déploiement de chaque **Distributed CVM**. Le **serveur global RMI** permet d'exécuter du code java d'une autre machine virtuelle.
Nous avons décidé de distribuer les composants dans deux **Distributed CVM**. Le premier représente le serveur et le deuxième représente le client
Le serveur contient le composant de controleur d'admission et les composants ordinateur. Le client contient le composant application container.

4.2 Lancement

Pour compiler le code, nous exportons via Eclipse le projet au format jar. Eclipse nous permet de compiler très simplement le code et nous évite de devoir écrire un makefile nécessitant de décrire les dépendances entre tous les fichiers.
Le code une fois compilé dans un fichier jar et placé dans le dossier jars, il faut alors se diriger dans le dossier bin.
Ensuite, via des terminaux pouvant exécuter des scripts bash, dans chaque terminal, il faut executer les commandes suivantes : ./start-cyclicbarrier config.xml ./start-gregistry config.xml ./start-controller 5 ./start-container APP0

4.3 Difficultés

Nous avons rencontré différents problèmes lors de l'exécution du code des différentes **Distributed CVM**.
Le soucis est qu'une erreur rmi est à notifier. Elle est déclenchée par le connecteur, lorsque nous essayons de faire transiter une admission d'une JVM à l'autre. Il se trouve que l'admission n'est pas sérializable alors que son interface le stipule. Il se trouve que la compilation ne détecte pas cette erreur.
Actuellement, le serveur et le client sont biens synchronisés. Le déploiement se déroule sans soucis. L'erreur survient lors de l'étape start.