CONTRIBUTION A L'IMPLEMENTATION D'UN MODEL MINIAML DE COMPOSANT

PSTL

Responsable du projet : Jacques Malenfant

Auteur : Alexander Annic

01 JANVIER 2015

Table des matières

1 Introduction

La première année de Master à l'UPMC comprend la réalisation d'un projet à choisir parmi une liste proposée par les enseignants de l'université. Ces projets s'étendent sur un semestre entier et constituent un parallèle avec la formation dispensée.

La programmation concurrente et répartie est un concept que je n'ai pas eu l'occasion d'étudier durant mes précédentes années d'études. C'est un sujet qui m'intéresse particulièrement car il s'avère fondamental dans l'informatique mondialisé d'aujourd'hui. Par conséquent, j'ai saisi l'opportunité que présentait ce travail pour aborder concrètement ce sujet.

Le projet choisi consiste à contribuer à l'implantation d'un modèle académique développé dans le cadre de la recherche : le *Basic Component Model* (BCM). Le BCM est une application à base de composants répartis en Java utilisant principalement des technologies de bas niveau tel que les sockets et les threads mais également l'API Java RMI[1].

Dans un premier temps, je décrirai certains mécanismes du BCM. J'exposerai ensuite les problématiques du projet pour enfin détailler sa réalisation.

[1] RMI *(Remote Method Invocation)*, technologie développée par *Sun* pour mettre en œuvre facilement des objets distribués.

2 Modèle minimal de composants

Le BCM permet de construire des applications réparties en Java contenant plusieurs milliers de composants déployés sur des dizaines de machines virtuelles. Il est développé par le professeur Jacques Malenfant, enseignant-chercheur à l'UPMC et est utilisé dans le cadre d'un projet de recherche ainsi que comme outil pour les travaux pratiques de l'UE ALASCA du Master 2.

Ce chapitre n'a pas pour objectif de décrire le fonctionnement de l'application mais seulement d'introduire certains points nécessaires à la compréhension du projet.

2.1 Registre global

L'application utilise la technologie RMI afin notamment de lier les assemblages de composants entre eux. Cette technologie a pour politique d'interdire la modification d'une entrée qui aurait été ajoutée depuis une machine distante. Cela implique qu'une entrée doit être publiée sur le même ordinateur que celui ayant effectué la requête. Pour contourner cette limitation, chaque machine exécute son propre registre RMI. Les composants publient ainsi les objets localement. Cependant, lorsqu'un composant cherche à se connecter à un objet distant, il a besoin de savoir sur quel registre RMI se connecter.

Le registre global est utilisé comme un répertoire regroupant les informations associant les entrées aux registres qui les hébergent. Ces informations sont stockées dans une table de hachage.

2.2 Déploiement réparti

Dans le cas d'un déploiement réparti de l'application, un fichier de configuration écrit en XML regroupe les informations nécessaires pour lier les différents assemblages de composants entre eux. Les lignes ci-dessous présentent un tel fichier pour une application simple mettant en action deux composants répartis sur deux machines physiques (192.168.0.14 et 192.168.0.12).

```
1.  <deployment>
2.      <cyclicBarrier hostname="192.168.0.14" port="55253"/>
3.      <globalRegistry hostname="192.168.0.12" port="55252"/>
4.      <rmiRegistryPort no="55999"/>
5.      <jvms2hostnames>
6.          <jvm2hostname jvmuri="provider"
7.                        rmiRegistryCreator="true"
8.                        hostname="192.168.0.12"/>
9.          <jvm2hostname jvmuri="consumer"
10.                       rmiRegistryCreator="true"
11.                       hostname="192.168.0.14"/>
12.     </jvms2hostnames>
13. </deployment>
```

On distingue quatre composants. Le composant *cyclic barrier* qui gère la synchronisation entres les autres composants (ligne 2), le registre global décrit précédemment (ligne 3) et les composants *provider* et *consumer*. Un registre RMI est créé par chacun des composants *provider* et *consumer* de manière à ce qu'il y en ait un par machine physique.

La figure 1 présente l'organisation de l'application conformément au fichier XML. Les flèches illustrent les étapes de communication entre un composant et les registres RMI. Celles en vertes concernent l'ajout d'une entrée dans un registre RMI et celles en rouges indiquent la modification d'une entrée depuis une machine distante. La communication entre les composants et le registre global se fait par l'intermédiaire d'un client. La classe implémentant ce client est appelé *GlobalRegistryClient*.

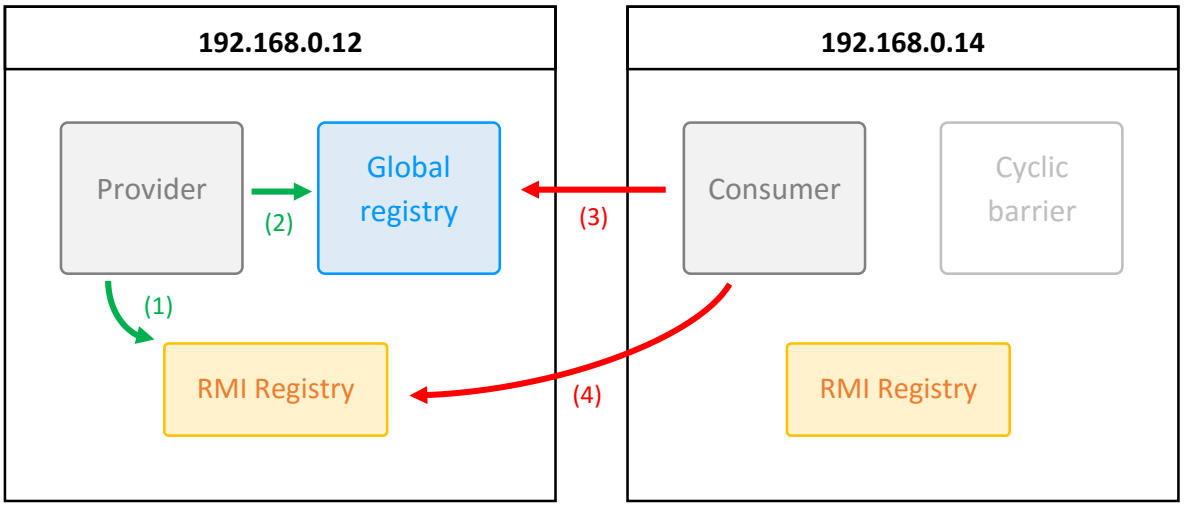


Figure 1 - Schéma décrivant le fonctionnement de l'application pour un cas simple

(1) : Le composant *provider* ajoute une entrée dans son registre RMI local.

(2) : Il prévient le registre global qu'une entrée a été ajoutée au registre RMI présent sur sa machine.

(3) : Le composant *consumer* souhaite effectuer une modification sur une entrée. Il consulte le registre global pour connaître l'adresse du registre RMI détenant cette clé.

(4) : Une fois cette adresse récupérée, le composant peut se connecter au registre RMI pour y modifier l'entrée.

2.3 Protocol de communication

La communication entre le registre global est un composant suit un protocole relativement simple décrit ci-dessous.

La flèche $\Rightarrow$ est utilisée pour indiquer une communication allant du composant au registre distribué. A contrario, la flèche $\Leftarrow$ indique une communication du registre vers un composant.

⇒ **put *key value***
Ajout d'une entrée $key \rightarrow value$ au registre.
⇐ **ok**
⇐ **nok**

⇒ **remove *key***
Suppression d'une entrée du registre.
⇐ **ok**
⇐ **nok**

⇒ **lookup *key***
Recherche d'une clé. Si la clé est présente, retourne sa valeur.
⇐ **ok *value***
⇐ **nok**

⇒ **shutdown**

3 Problématique

3.1 Automatisation du démarrage

Le lancement de l'application est une opération fastidieuse. En effet, chaque composant doit être lancé manuellement sur les différentes machines virtuelles et ordinateurs physiques. Il est également nécessaire que les sources compilées de l'application soient présentes sur les différents ordinateurs pour y démarrer un composant.

La première partie du projet consiste à développer un script automatisant cette procédure.

3.2 Améliorer le registre de publication global

Il arrive que l'application soit divisée en plusieurs milliers de composants répartis sur plusieurs dizaines de machines virtuelles. Par conséquent, le registre global peut être amené à faire face à un nombre trop important de requêtes ce qui peut alors perturber le fonctionnement de l'application.

La deuxième partie du projet consiste à répartir le registre global sur différentes machines afin d'éviter ce phénomène de "goulot d'étranglement". La répartition du registre doit être effectuée de façon transparente de manière à ne pas avoir à modifier l'implémentation des composants.

Cette partie représente l'objectif principal du projet. Elle sera, par conséquent, beaucoup plus détaillée que la première.

4 Déploiement

4.1 Principe

L'objectif est de développé un script simplifiant au maximum le lancement de l'application.

Dans un premier temps, il est nécessaire d'analyser le fichier de configuration XML afin de récupérer les informations utiles au déploiement. Il faut parcourir ensuite l'ensemble des machines physiques présentes dans ce fichier et vérifier que les sources de l'application soient présentes. Enfin, les composent doivent être lancé sur leur machine respective et dans le bon ordre. Par exemple, il est nécessaire que le composants *cyclic barrier* soit démarré avant les autres.

4.2 Choix d'implémentation

Perl s'est imposé comme étant le langage idéal pour répondre à cette problématique d'une manière simple et efficace possible. En effet, l'analyse du fichier XML peut facilement être réalisée en utilisant le moteur des expressions régulières natif à Perl, ce permet de ne pas utiliser de module externe. Le transfert et l'exécution des sources peuvent aisément être effectués par l'utilisation des commandes *Shell scp* et *ssh* permettant respectivement de copier des fichiers et d'exécuter des commandes sur une machine distante.

Par ailleurs, Perl est également accessible depuis Windows. Actuellement, le script réalisé n'est pas compatible avec ce système d'exploitation, notamment à cause de l'utilisation des commandes *scp* et *ssh*. Cependant, en remplaçant ces commandes par d'autres technologies adaptées, il est tout à fait possible de rendre le script exécutable sous Windows.

En conclusion, le langage Perl a permis de développer facilement un script autonome (contenu dans un seul fichier) qui ne soit soumis à aucune dépendance externe.

Pour automatiser le déploiement de l'application, il est nécessaire de disposer d'informations supplémentaires autres que celles définies dans le fichier de configuration XML :

- Le transfert et l'exécution étant réalisés via *ssh*, il est nécessaire de connaître le nom de compte de la machine sur laquelle effectuer ces actions;
- Le *classpath* des différents composants que l'on souhaite exécuter;
- L'emplacement des sources dans le cas où le script ne serait pas situé à la racine du projet;
- Le chemin indiquant l'endroit où transférer les sources sur la machine distante;
- Les différents fichiers que l'on souhaite transférer. A noter que les sources seules ne suffisent pas à lancer un composant. En effet, ce dernier a notamment besoin d'un fichier de type *policy*, d'un fichier définissant la grammaire du fichier de configuration XML ainsi que l'ensemble des *jars* nécessaire à l'application.

Bien que cela soit possible, il n'est pas pratique de définir l'ensemble de ces arguments en ligne de commande, car ils sont relativement nombreux. Par conséquent, si aucun argument n'est passé en ligne de commande, le script cherchera dans son répertoire courant un fichier, nommé par défaut "properties", regroupant ces informations. Le fichier "properties" a une structure de la forme $clé = valeur$. Son analyse par le script se fait alors très facilement par l'utilisation d'une expression régulière.

5 Registre global distribué

L'objectif est de répartir le registre global sur plusieurs machines. On divise ainsi, en moyenne, le nombre de requêtes que doit absorber le serveur par le nombre de machines disponible. La difficulté consiste à trouver une structure qui soit à la fois performante et robuste.

5.1 Ingénierie

5.1.1 1ère proposition

Une première possibilité relativement simple à mettre en place mais peu ambitieuse consiste à démarrer, au lancement de l'application, un registre sur chacune des machines physiques disponibles. Les clés sont alors distribuées sur les registres de manières autonomes ce qui assure une répartition parfaitement équitable dans la mesure où le phénomène de "goulot d'étranglement" que l'on cherche à éviter ne risque de se produire que si un nombre importants de clés a été ajouté aux registres. Les composants parcourent ensuite l'ensemble des registres, dans un ordre aléatoire pour éviter de surcharger les premiers registres, jusqu'à trouver la clé souhaitée.

Avantages :

- Facile à implémenter;
- Distribution idéale des clés sur les registres.

Inconvénients :

- Création du nombre de maximal de registres sans que cela ne soit forcément nécessaire;
- Complexité de recherche d'une clé en $O(n)$ ce qui rend l'application peu performante. Par ailleurs, cela signifie qu'une seule requête est amenée à être répété plusieurs fois. Par conséquent, la diminution du nombre de requête par registre n'est pas optimale.

5.1.2 2ème proposition

La seconde proposition permet de répondre aux deux inconvénients engendrés par la précédente. Elle s'apparente à une liste chaînée. Un registre, auquel est défini un nombre de clés maximal, est initialement démarré sur l'une des machines physiques. Lorsque ce maximum est atteint, le registre crée un semblable sur la prochaine machine définie dans le fichier XML et garde en mémoire l'adresse de cette dernière. Cela nécessite qu'un registre ne puisse se diviser qu'une seule fois. En effet, un registre ne connait que le registre qu'il a lui-même créé. Par conséquent, il n'est pas en mesure de connaître les machines disponibles pour se diviser une fois de plus. Pour s'efforcer de préserver un équilibre dans la répartition des clés, le premier registre conserve les $\frac{1}{n-k}$ première clés, où n représente le nombre de registre et k le nombre de registres précédemment créés.

Avantages :

- Adaptabilité aux besoins du nombre de registres

- Dans le pire cas, la complexité de recherche est en $O(n)$, car il faut parcourir tous les registres pour trouver la clé recherchée. Cependant, tous les registres ne sont pas forcément créés. De plus, le client garde en mémoire le registre déjà consulté. La complexité en moyenne est donc nettement meilleure que pour la première proposition.

Inconvénients :

- Les premiers registres se trouvent beaucoup plus sollicités que leurs successeurs;
- La répartition des clés n'est pas forcement équitable

5.1.3 3ème proposition

La dernière proposition fonctionne avec deux niveaux de registres.

Cette proposition met en place en "registre des registres". On appellera dorénavant registres distribués les registres contenant les informations associant les entrées des RMI avec les URI et registre global le "registre des registres" qui contient la liste des registres distribués associés à leur couverture d'entrées.

Au lancement de l'application, le registre global et un registre distribué couvrant les clés allant de A à Z sont créés. Lorsque le nombre de clés que contient le registre distribué dépasse une limite prédéfinie, ce dernier demande au registre global l'adresse d'une machine disponible. Il créé alors un nouveau registre distribué sur cette machine et y transfert la moitié de ses clés. La couverture des clés est alors actualisée en fonction de cette division. Chaque fois qu'un nouveau registre distribué est créé, il s'inscrit auprès du registre global. Le registre global garde ainsi une liste exhaustive des registres distribués. A noter que lorsqu'il n'y a plus de machine disponible, la limite de nombre d'entrées maximal ne tient plus. La figure 2 illustre la division d'un registre dont la limite des entrées serait fixée à 4 éléments.

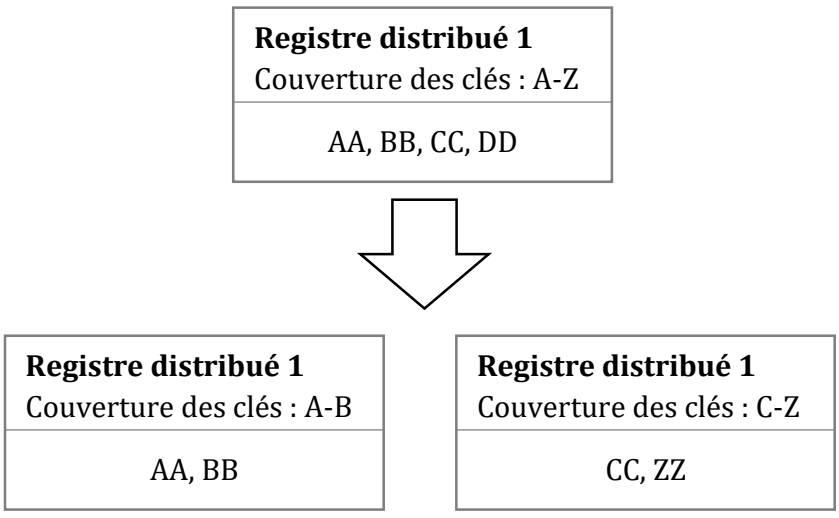


Figure 2 - Schéma illustrant la division d'un registre

Lorsqu'un composant cherche une entrée dans le registre, le client qui lui est associé consulte sa liste interne pour y trouver le registre distribué supposé contenir cette entrée. Le client interroge alors ce registre. Si ce registre ne détient pas l'entrée (parce qu'il s'est divisé),

il interroge le registre global. Celui-ci répond avec l'adresse du registre détenant l'entrée souhaitée.

Pour éviter de répéter plusieurs fois la même requête, les registres distribués et les composants (par l'intermédiaire de leur client) enregistrent en interne les informations concernant les registres distribués récupérés. Ces informations sont enregistrées dans une table de hachage. La figure 2 résume le fonctionnement de ce registre à deux niveaux.

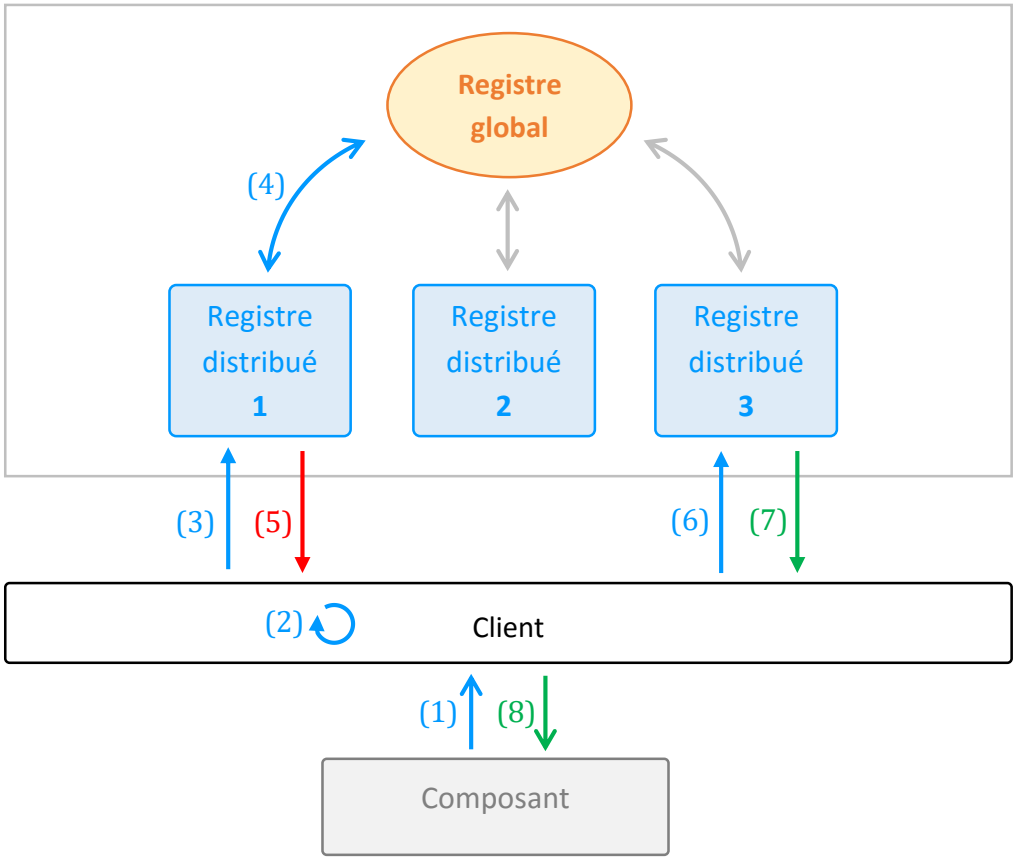


Figure 3 - Schéma illustrant la recherche d'une entrée par un composant dans le registre de la 3ème proposition

(1) : Un composant souhaite effectuer une opération sur l'ensemble des registres distribués (ajout / suppression / recherche). Supposons qu'il souhaite ajouter une clé x. Il interroge le registre par l'intermédiaire de son client.

(2) : Le client recherche dans sa table interne l'adresse du registre distribué susceptible de couvrir x.

(3) : Il interroge le registre n°1 qui d'après les informations contenues dans sa table devrait couvrir x.

(4) : Le registre n°1 ne couvre pas x (parce qu'il s'est divisé et que le client n'a pas effectué de requête sur celui-ci depuis). Ce registre interroge alors le registre global qui lui retourne l'adresse du registre n°3. Il actualise sur sa table interne ces nouvelles données afin de ne plus avoir à interroger le registre global si un autre composant venait à formuler une requête similaire.

(5) : Le registre n°1 retourne à son tour l'adresse du registre n°3 et actualise également sa table de données.

(6) (7) (8) : Le client transmet maintenant sa requête au registre n°3. Ce dernier ajoute x à son répertoire et confirme cet ajout auprès du client. A son tour, le client confirme au composant l'ajout de la clé x.

Avantages :

- Complexité de recherche d'une clé en $O(1)$;
- Répartition des clés optimale**.**

Inconvénients :

- Besoin de créer et de tenir à jour un registre supplémentaire

5.2 Verdict

La première proposition pose des problèmes de performances non négligeables. J'ai estimé avoir suffisamment de temps pour ne pas être contraint de la développer. Le choix était donc restreint entre les deux suivantes. La troisième solution est la plus performante des deux. De plus, la seconde a le défaut de ne pas assurer un équilibre en terme de nombre de clés entre les tous les registres. Par conséquent, le phénomène de goulot d'étranglement est plus probable dans celle-ci. Le seul inconvénient de la dernière proposition est l'ajout d'un registre globalisé. Cependant, dans une application à grande échelle, l'ajout d'un registre supplémentaire peut sembler négligeable. C'est pourquoi la troisième solution semble être plus avantageuse.

5.3 Réalisation

5.3.1 Nouveau protocole de communication

- **Registre global ↔ Registre distribué**

  ⇒ **seekHost *x*-*y***
  Un registre distribué envoie cette commande lorsqu'il souhaite se diviser. x-y représente la couverture de clé du nouveau registre. Le registre global peut ainsi préparer l'ajout dans son répertoire de l'adresse d'un nouveau registre auquel est associée cette couverture.

  ⇐ **ok *hostname***
  Si une machine physique est disponible, le registre global retourne son adresse.

  ⇐ **nok**
  Cette réponse est émise s'il n'y a plus de machine physique disponible.

  ⇒ **seekKey *x***
  Si un composant émet une requête à un registre alors que ce dernier ne détient pas la clé associé à cette requête, alors le registre envoie ce message au registre global.

⇐ **ok *hostname x-y***
Le registre global renvoie l'adresse du registre ainsi que sa couverture de clé afin que le registre distribué puisse l'inscrire dans son répertoire.

⇐ **nok**
Ce message est retourné si aucun registre ne couvre la clé x. Ce cas ne devrait jamais se produire.

⇒ **register *x-y***
Un registre distribué nouvellement créé envoie ce message pour confirmer sa création auprès du registre global. La couverture x-y est envoyée pour que le registre global puisse déterminer le registre concerné dans le cas où plusieurs registres distribués seraient ne seraient pas confirmés.

⇐ **ok**
La confirmation a réussie.

⇐ **nok**
Le registre global n'avait aucun registre distribué en attente de confirmation pour la couverture x-y dans son répertoire. Là encore, ce cas ne devrait jamais se produire.

⇒ **shutdown**
Déconnexion spontanée du registre distribué.

- **Registre distribué ↔ composant**

Ce protocole est similaire au protocole entre la version simple du registre anciennement implanté et un composant. La différence est la réponse **sync *x-y hostname a-b*** du registre lorsqu'il ne détient pas la clé passée en paramètre. Le client peut ainsi se mettre à jour pour ne pas réitérer une requête similaire.

put *key value*

⇐ **ok**
⇐ **sync *x-y hostname a-b***
⇐ **nok**

⇒ **remove *key***
⇐ **ok**
⇐ **sync *x-y hostname a-b***
⇐ **nok**

⇒ **lookup *key***
⇐ **ok *value***
⇐ **sync *x-y hostname a-b***
⇐ **nok**

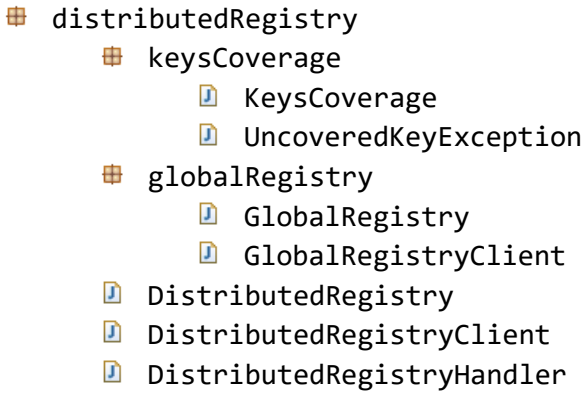
⇒ **shutdown**

5.3.2 Séparation des entrées

Afin de séparer les entrées d'un registre équitablement

5.3.3 Développement

Le nouveau registre est construit de la façon suivante :



- **KeysCoverage**

Cette classe a pour principal objectif de représenter la couverture de clés d'un registre. Elle possède plusieurs fonctions utilitaires dont la plus notable est celle permettant de définir la séparation d'un ensemble de clés.

- **DistributedRegistry**

Cette classe implémente le registre distribué. A son lancement, le registre écoute les requêtes sur un port donné. Lorsqu'un composant se connecte, un nouveau thread lui est assigné. Ce thread est chargé de gérer les requêtes du composant qui lui est associé jusqu'à ce que ce dernier se déconnecte.

- **DistributedRegistryClient**

Cette classe permet de communiquer facilement avec un registre. Elle associe un socket réalisant la connexion entre elle-même et le registre avec une couverture de clé.

- **DistributedRegistryHandler**

Du fait qu'il y ait plusieurs registres, un composant doit posséder un ensemble de clients (un pour chaque registre). Ainsi, chaque composant possède une instance de cette classe. Cette dernière possède une liste de *DistributedRegistryClient*. Lorsqu'un composant effectue une requête, cette classe parcourt la liste de ses clients afin de déterminer lequel est susceptible d'y répondre positivement. Si le registre ne détient pas la clé, il répondra avec l'adresse d'un nouveau registre. Un nouveau client sera alors ajouté à la liste de cette classe permettant de dialoguer avec ce registre.

- **GlobalRegistry**

Implémentation du registre global. Tout comme pour la classe *DisitributedRegistry*, cette classe écoute les requêtes sur un port prédéfini. Pour chaque registre distribué, un thread chargé de communiquer avec ce dernier est créé. La connexion est maintenue jusqu'à la fin du déroulement de l'application.

- **GlobalRegistryClient**

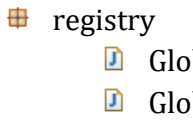
Simple client permettant de communiquer avec le registre distribué.

- **UncoveredKeyException**

Exception lancée par un client lorsqu'aucun registre défini dans son répertoire ne couvre une clé recherchée.

5.3.4 Intégration

Le registre simple anciennement développé est principalement composé de deux classes :



L'objectif est rendre un composant interopérable de manière à ce qu'il puisse facilement alterner entre ce registre simple et la nouvelle version distribuée en modifiant le moins possible sa définition.

Un composant doit implémenter la classe abstraite *AbstractDistributedCVM* afin d'hériter des propriétés nécessaires. Dans l'implémentation originale de l'application, cette classe contient actuellement une instance de la classe *GlobalRegistryClient*. Une solution est alors de remplacer cette instance par la définition d'une interface du même nom définissant le comportement souhaité du client. L'instanciation du client se fait alors au niveau de la définition concrète du composant.

Afin de factoriser les comportements communs entre les deux versions du registre, la classe *GlobalRegistry* devient une classe abstraite dont hérite la classe *SimpleRegistry* et *DistributedRegistry*.

Concrètement l'organisation du *package* est modifiée comme présenté ci-dessous. L'implémentation du simple registre est déplacé dans le *package registry.simpleRegistry* comme présenté ci-dessous et le registre distribué est situé dans le *package registry.distributedRegistry*.

- registry
  - simpleRegistry
    - SimpleRegistry
    - SimpleRegistryClient
  - distributedRegistry
    - ...
  - GlobalRegistry
  - GlobalRegistryClient

La figure **X** présente l'architecture des deux versions des registres sous la forme d'un diagramme UML. Les liens doubles de couleurs bleus (=) modélisent la communication entre les différents composants à travers des sockets conformément aux protocoles définis **X** pour le registre simple et **X** pour le registre distribué.

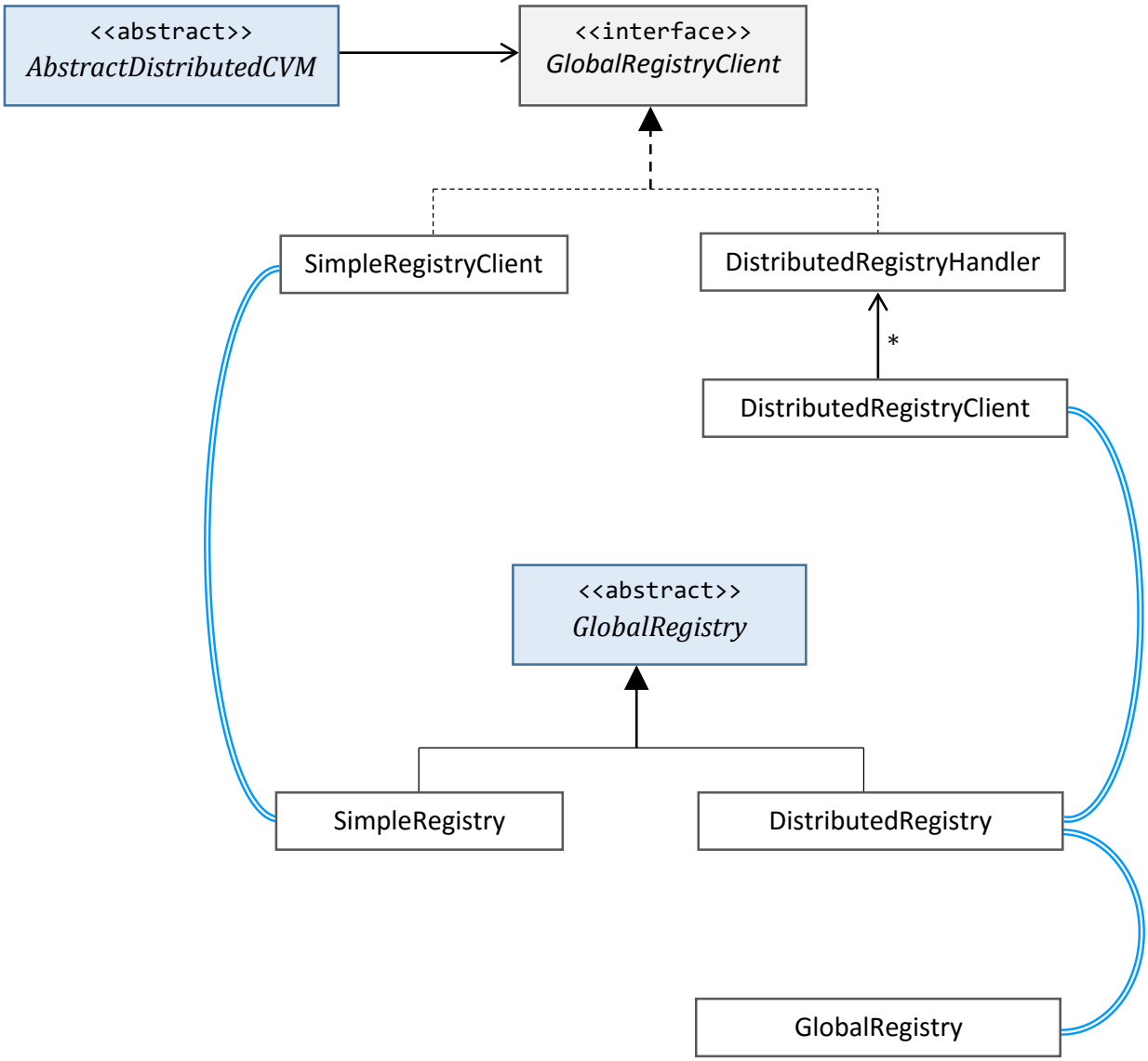


Figure 4 - Représentation UML de l'architecture des registres

5.4 Exécution

Pour visualiser concrètement les échanges ayant lieu entre les différents clients et les registres, je simule le comportement des clients en utilisant le programme *telnet*. La simulation présente un cas classique de fonctionnement des clients et des registres et ne couvre pas toutes les configurations possibles. Il ne s'agit pas d'un jeu d'essais.

5.4.1 Communication avec le registre global

```
>telnet localhost 55353
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
1.  seekHost ca-z
2.  ok 192.168.1.21
3.  register ca-z
4.  ok
5.  seekKey a
6.  ok 192.168.1.25 a-c
7.  seekKey x
8.  ok 192.168.1.21 ca-z
9.  seekHost fa-z
10. nok
```

Répertoire initial

192.168.1.21		null	false
192.168.1.25		a-z	true

Répertoire après seekHost *(ligne 1)*

192.168.1.21	ca-z	false
192.168.1.25	a-z	true

Répertoire après register *(ligne 3)*

192.168.1.21	ca-z	true
192.168.1.25	a-c	true

- **Etat initial**

Initialement, le tableau contient la liste des adresses des machines physiques. Cette liste est récupérée depuis le fichier de configuration XML. On identifie le registre créé au lancement de l'application sur la machine 192.168.1.25. Il couvre bien évidemment toutes les valeurs de clés possibles.

- **Ligne 1 et 2**

Le registre distribué souhaite se diviser, typiquement parce qu'il a atteint son nombre maximal de clés. Il demande au registre global l'adresse d'une machine libre. Le registre global associe la couverture des clés passée en argument à une adresse de machine libre qu'il retourne au client. A ce moment, la couverture des clés du premier registre n'est pas modifiée car le nouveau registre n'a pas encore confirmé sa création.

- **Ligne 3 et 4**

Le nouveau registre distribué confirme sa création. Le registre global met à jour son répertoire en définissant ce registre comme actif et en modifiant la couverture du premier registre.

- **Ligne 5 à 8**

Le client cherche les registres contenant les clés a et x. Le registre global répond conformément à son répertoire.

- **Ligne 9 et 10**

Le client cherche une nouvelle machine pour se diviser. Le registre global répond par nok afin d'indiquer qu'il n'y en a plus.

5.4.2 Communication avec un registre distribué

Cette simulation cherche à exhiber la division d'un registre lorsque la limite d'entrée est atteinte. On limite ici à 3 le nombre d'entrées dans le répertoire du registre distribué.

- **Depuis 192.168.1.25**

```
>telnet localhost 55353
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
1.  lookup key
2.  nok
3.  put keya valuea
4.  ok
5.  put keyb valueb
6.  ok
7.  put keyc valuec
8.  ok
9.  lookup keyc
10. ok valuec
11. put keyd valued
12. ok
13. lookup keya
14. ok valuea
15. lookup keyc
16. sync a-keyb 192.168.1.21 keyba-z
```

- **Depuis 192.168.1.21 après division**

```
>telnet localhost 55353
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
17. lookup keya
18. nok
19. lookup keyc
20. ok
```

- **Ligne 1 à 10**

Ajout de 3 premières clés. La recherche d'une clé fonctionne comme attendu.

- **Ligne 11 à 16**

L'ajout d'une 4ème clé va engendrer la division du registre. Ainsi lorsque l'on recherche l'entrée *keyc*, le registre retourne l'adresse du registre la contenant. On y retrouve également les couvertures des deux registres permettant au client de mémoriser l'état des deux registres.

- **Ligne 17 à 19**

Comme attendu, le nouveau registre détient l'entrée *keyc* mais ne détient pas *keya*.

6 Conclusion

La complexité croissante des systèmes informatiques et leur évolution de plus en plus rapide suscitent un intérêt accru pour le développement à base de composants. De plus, la programmation répartie est, aujourd'hui, devenue un fondement de l'informatique. Ce projet a constitué une expérience très enrichissante en abordant de façon concrète ces deux aspects. Il m'a également permis de solidifier mes compétences Java en travaillent sur une application d'envergure.

Bien que les objectifs soient atteints, j'aurais souhaité disposer d'avantage de temps pour améliorer mon travail. En effet, certaines améliorations peuvent encore être apportées, comme par exemple améliorer le registre afin que l'application reste fonctionnelle malgré que l'un des registres distribués tombe en panne.