Rapport de Travail d'Études et de Recherche

Intégration d'Amazon EC2 dans un Resource Manager de la librairie ProActive

par PARISY Anthony et HAYECK Ibrahim

Encadré par : AMEDRO Brian et CAROMEL Denis

Sommaire

I) Introduction....4
II) Problématique....5
III) La librairie ProActive et son Scheduler....6
III.1) Framework en général....6
III.1.a) Présentation....6
III.1.b) Les objets actifs....6
III.1.c) Sémantique des communications....7
III.1.d) Déploiement des applications ProActive....8
III.1.e) Concept de Virtual Node....8
III.2) Le Scheduler de ProActive....10
III.2.a) Les notions de Scheduling dans ProActive :....10
III.2.b) Architecture globale du Scheduler....11
IV) Amazon....12
IV.1) EC2....12
IV.2) S3....15
V) Travail effectué....15
V.1) Sur EC2 15
V.1.a) Réservation et démarrage d'une instance....16
V.1.b) Lancement d'un noeud sur EC2....16
V.1.c) Ajout du noeud dans un Resource Manager déjà lancé....17
V.2) Sur S3 17
VI) Problématique et différentes solutions....18
VI.1) Étude des différentes solutions....19
VI.1.a) Tunnel SSH....19
VI.1.b) rmissh....20
VI.1.b.1) Présentation....20
VI.1.b.2) Configuration du réseau....21
VI.1.b.3) rmissh : protocole de communication....22
VI.1.c) PAMR....23
VI.1.c.1) Vue d'ensemble de ProActive Message Routing....23
VI.1.c.2) Configuration du ProActive Message Routing....24
VI.1.c.3) Message Router Configuration :....24
VI.2) Solution retenue : PAMR....24
VII) Perspectives....26
VIII) Remerciements....27
IX) Bibliographie....28
X) Annexes 29

Ce rapport a été écrit dans le cadre du TER[1] d'Ibrahim Hayeck et Anthony Parisy. Il a été réalisé au sein de l'équipe OASIS[2] à l'INRIA[3] de Sophia Antipolis, sous l'encadrement de Messieurs Denis Caromel et Brian Amedro.

I) Introduction

A l'heure où les ordinateurs deviennent de plus en plus puissants, la puissance de calcul devient, plus que jamais, une **ressource** à part entière.

Les entités nécessitant une forte puissance de calcul l'ont bien compris : la loi de Moore vit ses dernières heures, et l'avenir réside en ce que l'on appelle des **grilles de calcul**, ou *Grid*, qui consistent en un agrégat de machines "classiques" afin de constituer, à moindre coût, un super-calculateur.

Mais qu'en est-il des entités ne nécessitant pas nécessairement une telle puissance de calcul ? Ou tout du moins, pas de manière constante ? C'est dans ce but précis que le *Cloud Computing* s'est développé.

Dans ce rapport, nous allons vous présenter le service d'Amazon qui propose depuis quelques temps (sortie officielle le 23 Octobre 2008, mais bêta disponible depuis 2007) des offres lui permettant à lui (vendeur) de rentabiliser et d'entretenir ses infrastructures coûteuses, et à nous (consommateurs) d'obtenir à moindre coût des noeuds de calcul.

Dans le but de mieux vous exposer notre problématique, nous présenterons tout d'abord le cadre de travail dans lequel nous fonctionnions.

Nous présenterons donc d'abord la plate-forme pour le calcul distribué sur laquelle était basé ce TER (ProActive Parallel Suite), et plus précisément les parties dont nous nous sommes servies (Ressource Manager et Scheduler).

Puis, nous vous présenterons l'étude que nous avons réalisée de la plate-forme d'Amazon : EC2[4].

Nous expliquerons ensuite les différents problèmes et solutions que nous avons mis en oeuvre durant notre séjour à l'INRIA.

Nous finirons par détailler les différentes perspectives de ce TER.

---

1 Travaux d'Études et de Recherches

2 Objets Actifs, Sémantique, Internet et Sécurité

3 Institut National de Recheche en Informatique et en Automatique

4 Elastic Compute Cloud

Concernant la partie **gestion de projet**, la répartition des tâches fut rapide étant donné le nombre restreint de personnes. Nous avons toujours essayé de travailler en parallèle sur notre propre partie.

Ibrahim s'est chargé principalement des aspects de déploiement et de scheduling.

Anthony s'est plus focalisé sur les parties liés à Amazon (EC2 et S3).

Nous avons bien évidemment collaboré étroitement lors de la recherche de solutions aux différents problèmes que nous avons rencontrés. Nous avons d'ailleurs travaillé ensemble sur la partie concernant PAMR et les solutions d'intégration en général.

II) Problématique

L'objectif initial de ce TER était l'intégration automatique de noeuds[1] (au sens ProActive du terme) EC2 au sein d'un Resource Manager afin qu'ils soient correctement utilisés à des fins de calcul.

Cette intégration prend tout son sens lors de la surcharge de machines, ou tout simplement lors de l'approche d'une *deadline*, c'est à dire à un moment critique où il est nécessaire d'avoir le maximum de ressources disponibles pour pouvoir finir, à temps, le projet en question.

Il est donc nécessaire de pouvoir ajouter à un Scheduler existant des noeuds démarrés sur la plate-forme EC2. Mais cette hétérogénéité pose des problèmes de pare-feu car les noeuds n'appartenant pas aux mêmes réseaux, il ne peuvent pas necessairement communiquer de manière triviale.

Un "sous objectif" de ce TER était l'étude et la mise en place d'outils permettant de simplifier la procédure de mise à jour des AMI[2], utilisées par EC2. Plus précisément, on souhaiterait que, lors de la sortie d'une nouvelle version de la librairie de ProActive, l'image correspondante puisse facilement être mise à jour.

Maintenant, et afin de mieux comprendre toutes ces notions, nous allons vous expliquer en détail ce qu'est la librairie ProActive, mais également les différents *Web Services* proposés par Amazon.

1 Entité logique capable d'accueillir plusieurs objets actifs (définis par la suite)

2 Amazon Machine Image

III) La librairie ProActive et son Scheduler

III.1) Framework en général

Ce chapitre nous permet, dans un premier temps, d'introduire les notions de base de la programmation répartie plus spécialement le framework ProActive et ses propres concepts de Scheduling.

III.1.a) Présentation

ProActive est une bibliothèque Java pour le calcul parallèle, distribué et concurrent. Elle est conçue pour permettre de déployer une application sur un grand nombre de machines, et ce avec un maximum de transparence. Le langage Java a été retenu car il permet, avec sa machine virtuelle (JVM), une grande souplesse vis à vis des multiples architectures que l'on peut rencontrer sur une grille de calcul.

La bibliothèque permet la création d'objets distants, la mobilité des applications, les appels de méthodes asynchrones et des communications de groupe. Avec un ensemble réduit de primitives, ProActive fournit une API permettant la programmation d'applications distribuées pouvant être déployées aussi bien sur des réseaux locaux (LAN) que sur un ensemble de grilles de calcul inter-connectées via Internet.

L'un des avantages de cette plate-forme est qu'aucune modification de l'environnement d'exécution n'est requise, ni aucun pré-processeur ou compilateur spécial. Une machine virtuelle Java standard suffit à utiliser la bibliothèque. Le modèle de distribution de ProActive est parti d'un effort de simplification et d'un souci de réutilisation de code d'applications dans des systèmes à objets, en respectant une sémantique précise.

III.1.b) Les objets actifs

Les objets actifs sont l'un des concepts de base pour développer une application répartie avec ProActive. Un objet actif dispose de sa propre *thread* qui lui permet d'exécuter les méthodes invoquées sur cet objet actif par d'autres objets (actifs ou non). ProActive permet au programmeur de ne pas manipuler explicitement les *threads* et surtout toutes les dispositions nécessaires qui entourent leur utilisation.

Un objet actif peut être créé sur n'importe quel hôte utilisé pour le déploiement de l'application. Une fois cet objet créé, son activité et sa position (local ou distante) sont parfaitement transparentes.

Pour accueillir un objet actif, ProActive a introduit la notion de noeud qui permet d'identifier la JVM sur laquelle on souhaite le créer. Un noeud est une entité logique capable d'accueillir plusieurs objets actifs. Cependant, on préfère malgré tout conserver le modèle "un objet actif par noeud".
La création d'un objet actif se réalise alors comme suit :

```
MaClasse oa = (MaClasse) ProActive.newActive("MaClasse", parametres, noeud);
```

III.1.c) Sémantique des communications

Pour ces communications entre objets, ProActive s'appuie sur Java RMI[1]. Ce dernier a l'avantage d'être présent dans toutes les distributions standards de Java. Il est important de noter qu'un appel RMI est bloquant, ce qui peut entraîner des attentes inutiles dans l'exécution d'un programme, comme par exemple l'attente d'un résultat qui ne sera utilisé que plus tard.

Dans l'optique de parer à cet inconvénient de taille, ProActive permet d'effectuer des communications aussi bien asynchrones que synchrones. Le choix du mode de communication se fait en fonction de la signature de la méthode :

- Synchrone : la méthode retourne un objet non réifiable ou peut lever une exception. Dans ce dernier cas, l'appelant doit être bloqué pour éviter qu'il ne sorte du bloc {try/catch}. Au total, deux messages auront été échangés. L'un lors de l'appel de la méthode, et l'autre lors de la réception du résultat, une fois que la méthode aura été exécutée. Entre temps, la *thread* appelante est bloquée.

- Asynchrone : la méthode retourne un objet de type réifiable et ne peut lever aucune exception. Deux messages seront là aussi échangés, le premier pour l'appel de méthode et le second pour la réception du résultat. Mais, entre temps, la *thread* appelante n'aura pas été bloquée. Dans l'attente du résultat, ProActive génère un objet futur. Une utilisation de cet objet sera bloquante tant que le vrai résultat ne sera pas arrivé.

- Sens-unique : la méthode ne retourne rien (*void*) et ne lève pas d'exception. Un seul message est envoyé pour l'appel de méthode, et la *thread* n'est pas bloquée. Aucun futur n'est créé étant donné qu'il n'y a pas de résultat attendu.

Les objets futurs sont utilisés par ProActive pour créer un comportement asynchrone alors que les appels RMI sous-jacents sont eux entièrement synchrones. En effet, dès qu'un appel de méthode sur objet actif est réalisé, ProActive construit et retourne immédiatement un objet vide simulant l'objet attendu : le futur. La requête RMI est alors déléguée à un autre fil d'exécution. Une fois la requête traitée, le résultat obtenu est placé dans le futur. Le futur implémente la même interface que l'objet résultat.

Ce dernier point explique la nécessité pour un type retour d'être réifiable (du moins si l'on souhaite profiter de l'asynchronisme), car l'objet retourné devra pouvoir être sous-classé. Une classe finale ou un type primitif ne peut donc pas convenir.

Dans le cas où l'objet futur est utilisé alors que le vrai résultat de l'appel qui en est à l'origine n'est pas encore arrivé, ProActive introduit le mécanisme d'attente par nécessité qui bloque l'exécution jusqu'à l'arrivée du vrai résultat.

---

1 Remote Method Invocation

III.1.d) Déploiement des applications ProActive

Un des soucis majeurs de ProActive est de donner la possibilité d'écrire un code portable quant au déploiement de l'application. Pour cela, une procédure de déploiement est décrite dans deux fichiers XML :

- GCM Application Descriptor (GCMA) qui définit l'application à déployer. Il peut s'agir, par exemple, d'un natif C/C++ ou d'une application Java.
- GCM Deployment Descriptor (GCMD) qui indique où et comment l'application va être déployée. Cette partie définit les noms d'hôtes où se déroulera l'application, la manière (protocole) d'accéder aux ordinateurs (SSH, rsh, etc ...), les paramètres spécifiques au protocole d'accès à l'hôte, le chemin d'accès aux bibliothèques ProActive, etc ...

Dans le cas de Resource Manager, GCMA définit toujours le déploiement de noeuds ProActive. Cela signifie que, concernant le déploiement, la seule partie qu'un administrateur doit définir est le fichier GCMA représentant la description de l'infrastructure informatique.

III.1.e) Concept de Virtual Node

Le déploiement de ProActive est un concept abstrait qui est construit autour de nœuds virtuels. Au moment de l'exécution, un noeud virtuel sera composé d'un ou plusieurs noeuds, mais l'application n'a pas besoin de le savoir. Le déploiement se base sur les fichiers descripteurs cités précédemment.

Lors de son activation, le descripteur de déploiement instancie tous les *runtimes* ainsi que les noeuds nécessaires au fonctionnement de l'application. Il est alors possible de récupérer un objet représentant un noeud virtuel donné. Parmi les méthodes présentes dans la classe *VirtualNode*, la méthode *getNode()* permet de récupérer la référence sur un des noeuds contenu dans le noeud virtuel.

Exemple d'utilisation du descripteur de déploiement dans l'application (voir *Annexe X.IV)* pour détails) :

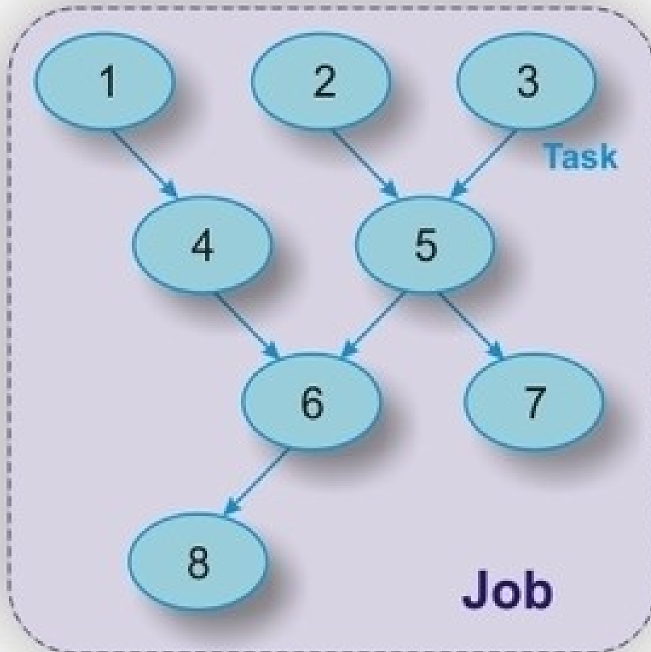
```
// arg[0] etant le descripteur de deploiement GCMD
GCMApplication deployer = PAGCMDeployment.loadApplicationDescriptor(new File(args[0]));
 //Activate all virtual nodes
 deployer.startDeployment();

 GCMVirtualNode vn = deployer.getVirtualNode("matrixNode");

Node a = vn.getANode();
m1 = (Matrix) org.objectweb.proactive.api.PAActiveObject.newActive(Matrix.class.getName(),
 parameters1,a);
```

III.2) Le Scheduler de ProActive

III.2.a) Les notions de Scheduling dans ProActive :



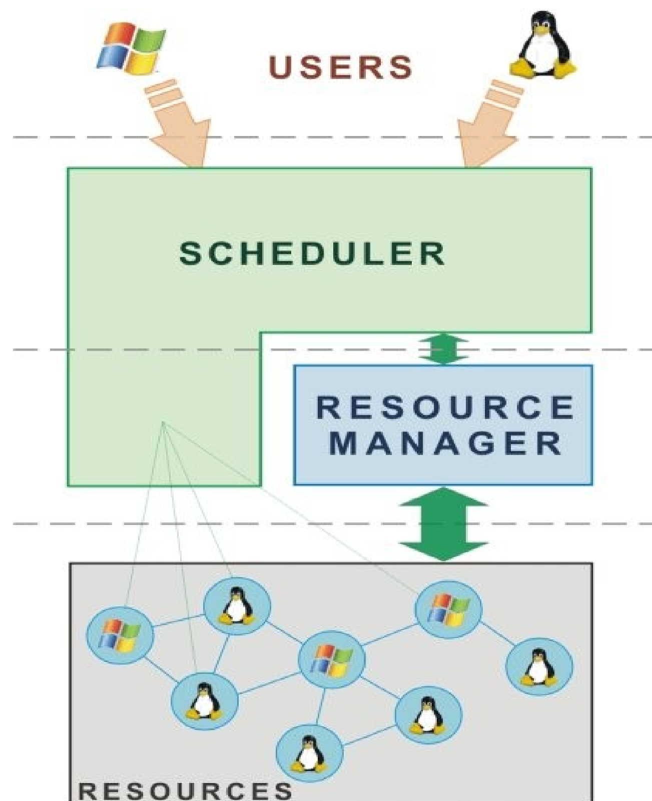
Avant toute chose, qu'est-ce qu'un job ?

Un job est l'entité qui sera soumise au Scheduler. Il est composé d'une ou plusieurs tâches. Il existe deux types de jobs, décrits ci-dessous :

- TasksFlow : représente un travail contenant un ensemble de tâches, qui peuvent être exécutées soit en parallèle, soit en fonction d'un arbre de dépendance. Les tâches dans ce travail peuvent être *Java* (un groupe écrit en Java, une extension de l'interface) ou *Native* (natif tout processus).

- ProActive : représente un travail qui contient une application ProActive (intégrée dans une seule tâche ProActive). Son exécution commence avec un nombre prédéfini de ressources, sur lesquelles l'utilisateur peut lancer l'application ProActive. Ce genre de travail exige l'API ProActive, afin de construire une application ProActive.

III.2.b) Architecture globale du Scheduler



Le Scheduler de ProActive est le fruit d'une collaboration entre deux entités principales : le Scheduler Core et le Ressource Manager. Chacune d'entre elles dispose de ses propres fonctionnalitées.

Le Scheduler Core est une interface qui est connectée au Resource Manager. Il est en charge de la planification des tâches soumises, conformément à la politique d'ordonnancement. Afin de lancer le job, le Scheduler se connecte à un Resource Manager pour obtenir des noeuds (ressources). C'est ainsi que l'utilisateur interagit avec l'entité Scheduler afin de déployer des tâches sur les noeuds fournis par le Ressource Manager. Remarque : le Scheduler de ProActive est un Scheduler multi-language, ce qui offre la possibilité à l'utilisateur de lancer des jobs contenant des tâches utilisant des exécutables C ou C++.

Soumettre un job au Scheduler revient à soumettre un flux (ou **ensemble**) de tâches. Une tâche peut être définie comme une étape d'un job. C'est la plus petite partie d'un job. Le Scheduler utilise les nœuds de calcul pour exécuter des jobs. Il lance l'exécution d'une tâche sur chaque noeud. Lorsque la tâche est terminée, il lance la suivante, et ainsi de suite.

Le Resource Manager de ProActive s'appuie sur le fichier GCMD pour déployer les nœuds de calcul. Cela implique que l'application pourrait être déployée n'importe où, sans modifier le code source. Par exemple, vous pouvez utiliser différents protocoles pour la création de la JVM nécessaire pour l'application, tels que rsh, SSH, Globus, LSF, etc ... Des exemples de fichiers GCMD et GCMA sont illustrés dans les parties Annexe X.V) et Annexe X.VI) et permettent d'avoir une idée plus précise sur la manière d'écrire des descripteurs de déploiement mais également la manière de les utiliser à l'interieur du code de l'application. Il existe de nombreuses façons d'ajouter des nœuds au Resource Manager soit avec le fichier descripteur de deploiement GCMD, soit à partir de l'adresse et le port sur lequel le noeud est demarré.

IV) Amazon

IV.1) EC2

EC2 est le web-service proposé par Amazon à tout les clients souhaitant obtenir de manière simple et rapide des machines sur lesquelles héberger leurs applications. Cette offre, principalement destinée aux entreprises, à été proposée pour la première fois aux États-Unis le 23 Octobre 2008, et est arrivée en Europe dernièrement (le 10 Décembre 2008). Elle a été lancée initialement pour rentabiliser et entretenir les infrastructures coûteuses qu'Amazon avait mis en place pour ses autres activités.

Dans la pratique, comment cela fonctionne-t-il ?

La solution d'Amazon (EC2) est de proposer un service de **virtualisation** utilisant leurs infrastructures physiques (imposantes et sécurisés, mais je reviendrais sur ce point).

Concrètement, lorsque l'on réserve une machine (plus précisément appelé **instance** dans le cas d'EC2), une machine virtuelle est démarrée directement sur une de leurs machines physiques.

Cette solution offre plusieurs avantages :

- Côté serveur :
  - possibilité pour Amazon de démarrer plusieurs machines virtuelles sur une seule machine physique (optimisation de l'utilisation des ressources des machines physiques, ce qui offre un meilleur rendement)
  - possibilité de proposer des machines de tout type, quel que soit le support hardware en dessous (ex : une "petite" configuration tournant sur une "grosse" machine)
  - simplicité de mise en oeuvre (les machines peuvent rester démarrées 24/24h et l'utilisateur démarre simplement ses instances quand il le désire (meilleure réactivité)
- Côté client :
  - possibilités de choix de l'environnement (windows, linux, etc ...) (j'y reviendrais plus tard)
  - différents moyens d'accès aux instances (SSH, W*eb Services* ...)
  - transparence totale : les machines virtuelles sont vues exactement comme des vraies machines pour le client
  - contrôle précis du type d'instance désiré (matériel virtualisé)

Ces machines virtuelles peuvent bien sûr être plus ou moins puissantes, ce qui permet de répondre à des besoins spécifiques divers en fonction du type de client (particulier ou entreprise).

Les instances étant des machines virtuelles, il est possible à tout instant de les stopper ou de les redémarrer, comme toute machine physique classique.

Il est également possible d'en démarrer à tout moment (moyennant finance) avec un délai de l'ordre de la minute pour que la machine soit pleinement utilisable.

Cela entraîne un atout de taille : ce que l'on appelle la *scalability*, ou **passage à l'échelle**. En effet, et du fait de la configuration rapide et simplifiée de l'infrastructure (démarrage/arrêt de machines quasi instantané à l'échelle de temps d'un serveur), il est possible à tout moment de changer la configuration physique sur laquelle tournent les applications que l'ont désire, et cela dans les deux sens.

Par exemple, un serveur de téléchargement de fichier faisant face à une affluence de téléchargements simultanés (sortie de la dernière bande annonce HD du dernier film à la mode par exemple) risque de **saturer** face à l'affluence de clients. Grâce à EC2, on peut en quelques secondes réserver d'autres machines qui permettront de **tenir la charge**.

D'un autre côté, si on dispose d'un nombre de machines important (toujours dans le même cas du serveur de téléchargement de fichier) mais que l'on se situe en heures creuses, par exemple à 3h du matin où l'affluence n'est pas importante, on ne désire pas forcément payer pour des machines n'étant pas ou peu sollicitées.

Là encore, grâce à la modularité d'Amazon, il est possible de stopper les instances que l'on n'utilise pas afin ... d'économiser de l'argent tout simplement. Cela permet de payer uniquement pour nos besoins.

Pour une petite entreprise, une start-up, ou même un particulier, c’est évidement très intéressant, car les coûts fixes d’hébergement peuvent alors être beaucoup moins importants que par rapports à d'autres solutions déjà existantes. Et un aspect également très intéressant pour ceux-ci est que le temps d’ajustement est très court par rapport à celui des solutions classiques. Cela permet une forte réactivité et une utilisation toujours optimale des infrastructures.

Mais les grandes entreprises ne sont pas en reste car il existe bon nombre d'instances (j'y reviendrai plus tard) dont certainement sont tout spécialement dédiées aux grandes entreprises nécessitant différents types de ressources (CPU, RAM, HDD, bande passante, etc ...).

Concernant l'aspect **virtualisation**, et comme je l'ai dit précédemment, il est possible de démarrer ses propres instances, correspondant à des machines virtuelles. Chacune de ces machines virtuelles dispose donc de son propre OS[1]. Cet OS peut être, au choix, l'un parmi ceux proposés initiallement par Amazon (OS classiques : windows server, fedora, debian, ubuntu, etc ...), soit l'OS que vous désirez.

Une autre aspect très important des instances est ce qu'Amazon appelle les AMI. Ces AMIs vous permettent, de manière simple et rapide, de **configurer** très finement le contexte dans lequel vos applications tourneront.

Elles peuvent contenir toute sorte de programmes, base de données et/ou librairies. Dans l'exemple de l'équipe OASIS, il existe une AMI qui contient l'ensemble de la librairie de ProActive.

1 Operating System : Système d'exploitation

Cela permet à l'utilisateur de très simplement démarrer des instances qui seront assurées d'être **compatibles** avec les applications utilisant vos librairies (dans le cas présent, ProActive). Cet aspect est non négligeable car la reconfiguration /installation de tout un environnement lors du démarrage de nouvelles instances (en cas de montée en charge par exemple) fait gagner un temps considérable : une simple configuration du (ou des) AMIs que vous voulez lier à une instance et elle démarrera avec la configuration désirée.

Il est également possible de lier plusieurs AMIs à une seule instance. Cela permet un contrôle encore plus fin du contexte applicatif. Par exemple, si on souhaite dissocier la librairie java de la librairie ProActive, il suffit de créer deux images. Ainsi, si une nouvelle version de Java sort, il est plus simple de mettre à jour l'image, plutôt qu'une grosse image contenant "tout".

Chacunes des AMIs que vous créez est liée à votre compte Amazon. Cela signifie qu'elles seront, à tout moment, TOUTES disponibles pour n'importe quel type d'instance que vous désirez.

Concernant ces différents types d'instances, Amazon propose, en fonction des besoins, différents types d'instances.Elles se divisent en deux catégories : les instances **standard** et les **High-CPU**.

La première catégorie regroupe des instances offrant un bon compromis CPU/RAM, ce qui permet à la plupart des applications de tourner sans problème.

La deuxième partie offre, comme son nom l'indique, beaucoup plus de puissance de calcul (CPU) que de RAM. Elles sont spécialement conçues pour des applications de type **calcul intensif**.

Chaque instance est facturée à l'heure, ce qui permet une modularité très fine (à l'échelle d'une entreprise). Cela permet par exemple de tester une application en production rapidement et à moindre coût, sur différentes configuration matérielles.

Je vous invite à regarder le tableau en *Annexe X.II)* pour de plus amples informations sur les différents types d'instances.

Un autre aspect important de la plate-forme Amazon EC2 est l'aspect **service**. En effet, EC2 est donné comme étant disponible 24/24h 7/7j avec un indice de confiance de plus de 99.95% ! Il en va de même pour la plate-forme S3[1] que je vais maintenant vous présenter.

1 Simple Storage Service

IV.2) S3

En parallèle avec EC2, Amazon a lancé sa propre plate-forme de stockage en ligne, appelé S3. C'est sur cette plate-forme que sont hébergées les différentes AMI liées au compte.

C'est cela qui permet, où que l'on soit, d'utiliser n'importe quelle configuration pour les instances EC2 que l'on souhaite démarrer.

Un des gros avantages de S3 est qu'Amazon se charge de tous les détails liés à la tolérance aux pannes (par exemple : Amazon prends en charge tout ce qui est redondance).

Cela signifie que les données que vous mettez sur leurs serveurs sont virtuellement **incorruptibles** et pourront y rester bien plus longtemps que sur un disque dur classique.

Cela simplifie également grandement la tache des administrateurs qui n'ont pas à s'occuper de tout les aspects **sauvegardes** et autres détails liés au renouvellement du matériel.

Je vais maintenant vous présenter le travail effectué sur ces deux outils.

V) Travail effectué

V.1) Sur EC2

La première étape de ce TER (outre la mise en place du cadre de travail : comptes gforge, configuration svn+ssh, etc ...) fut l'étude de la plate-forme EC2.

EC2 offre plusieurs manières d'accéder à ses services. L'une d'entres-elle est un ensemble d'outils en ligne de commande permettant de tester simplement et surtout rapidement des commandes basiques. C'est cette API[1] que j'ai utilisée le plus souvent durant ce TER.

Une autre approche se présente sous la forme d'un plugin firefox (Elasticfox) permettant, au sein d'une interface graphique, d'effectuer les mêmes actions basiques.

Que ce soit pour Elasticfox, les outils en ligne de commande, ou tout autre manière d'accéder aux web services d'Amazon, l'identification se fait de manière similaire : grâce à des fichiers de certification X.509.

Comme dit précédemment, j'ai simplement utilisé les outils en lignes de commande (ou *EC2 API Tools*) afin de réaliser mes tests. Pour de plus amples informations sur chacune des commandes, je vous invite à regarder en *Annexe X.I)* la partie concernée.

Le déroulement de l'étude d'EC2 et de la mise en application de ses outils s'est fait en trois étapes principales.

1 Application Programming Interface

V.1.a) Réservation et démarrage d'une instance

Pour commencer, j'ai tout d'abord utilisé la commande

```
ec2-desribe-images -a | grep OASIS
```

permettant de décrire l'ensemble des images AMI disponibles (première étape lors de la réservation d'une machine) appartenant à OASIS.

Une fois l'identifiant d'AMI récupéré, on peut l'utiliser pour lancer l'instance (ici c'est l'instance par défaut qui sera lancée, faute de spécification différente dans la configuration). Pour ce faire, on utilise la commande suivante :

```
ec2-run-instances XXX -k my-keypair
```

ou XXX représente l'identifiant de l'AMI à utiliser (ici une seule) et my-keypair représente la clé formée des fichiers de certification X.509.

Pour suivre l'avancement du démarrage de l'instance, mais également pour voir la liste des instances associées au compte, et qui tournent déjà, on utilise la commande

```
ec2-describe-instances
```

On voit alors que l'instance que l'ont vient de démarrer est d'abord *pending* (ce qui signifie qu'elle est en cours de démarrage). Puis, au bout d'une minute environ, elle passe en mode *running*, ce qui signale que l'instance est maintenant complètement accessible et utilisable. Remarque : pour terminer l'instance, on utilisera également un identifiant fourni lors de cette commande et on exécutera la commande suivante :

```
ec2-terminate-instances i-xxxxxxxxx
```

V.1.b) Lancement d'un noeud sur EC2

Une fois que l'instance est *running*, la commande *ec2-describe-instances* fournit le nom de la machine qu'il suffit ensuite d'utiliser pour effectuer une simple connection SSH à la machine.

Voici à quoi cela ressemble dans la pratique :

```
ssh -i my-keypair root@ec2-xxx-xxx-xxx-xxx.compute-1.amazonaws.com
```

Nous voyons ici que nous pouvons (et c'est normal) nous connecter en root sur la machine, et donc y faire ce que bon nous semble.

Dans le cas présent, on souhaite simplement lancer un noeud au sens ProActive du terme. Pour ce faire, et grâce à la librairie incluse dans l'AMI, il suffit de lancer la commande *startNode toto* afin de lancer un noeud nommé toto sur EC2.

V.1.c) Ajout du noeud dans un Resource Manager déjà lancé

Une fois le noeud lancé sur EC2, le plus gros du travail "semble" fait : on obtient une URL classique que l'ont peut ensuite ajouter au Resource Manager. Mais, et c'est la tout le problème, lors de l'ajout de cette URL dans un Resource Manager déjà lancé (et contenant un ensemble de noeuds **locaux**), la communication ne traverse pas le pare-feu. De ce fait, le noeud n'est pas ajouté, et donc jamais utiisé.

Pire, le Resource Manager attend une réponse du noeud EC2, réponse qui ne passera jamais le pare-feu, et donc bloquera tout le Resource Manager jusqu'à la fin du timeout.

Il est donc nécessaire de trouver des alternatives à ce problème, ou des moyens de contourner ce problème de pare-feu. (cf partieVI) Problématique et différentes solutions)

V.2) Sur S3

Une autre partie du TER fut consacré à la mise à jour des AMI contenant la librairie ProActive. La aussi, il existe plusieurs outils permettant de manipuler les AMI.

J'ai personnellement utilisé une implémentation en ruby appelé s3sync car, bien que non officielle, c'est l'implémentation de référence dans le domaine. En détail, s3sync utilise 3 variables pour s'associer à un compte : *AWS_ACCESS_KEY_ID*, *AWS_SECRET_ACCESS_KEY* et *AWS_CALLING_FORMAT (*qui doit avoir la valeur *SUBDOMAIN* dans le cas présent).

Une fois ces propriétés de sécurité exportées, on peut sans problème utiliser les différentes commandes liées à s3sync.

Première étape : on copie sur une machine EC2 le script "bundle_image.sh" (détaillé dans la partie *Annexe X.III)*) ainsi que les certifications X.509 (*.pem). C'est ce script qui fera tout le travail d'envoi sur la plate-forme S3.

Ensuite, on envoie ce que l'on désire grâce à la commande

```
s3sync.rb -r -v Proactive/dist/lib oasis-proactive-bucket:jars
```

Dans cet exemple, on envoie simplement tout les jars de la dernière version de ProActive. Une fois que tout les fichiers ont été synchronisés, on peut lancer le script "bundle_image.sh".

Une fois l'exécution de celui-ci terminée, il suffit de d'enregistrer l'image pour finaliser l'envoi, grâce à la commande

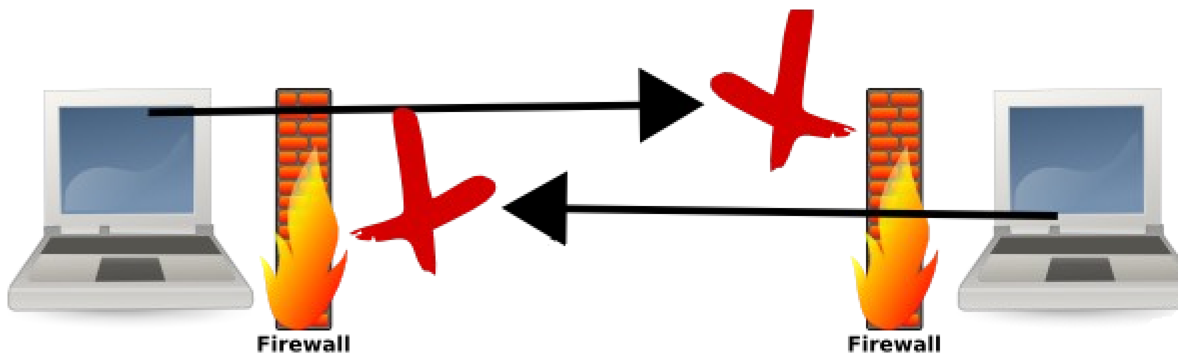
```
ec2-register oasis-proactive-bucket/proactive-image.manifest.xml
```

Cette commande se sert d'un manifeste (simple fichier XML) permettant de détailler le contenu de l'AMI (description, nom de l'image, propriétaire(s), etc ...).

VI) Problématique et différentes solutions

Le protocole par défaut de ProActive, RMI ouvre dynamiquement des connexions entre le client et le serveur pour faire passer ses communications, et c'est là qu'on s'est heurté au problème de pare-feu qui interdisait ce genre de communications.

RMI et les pare-feux :



Le problème principal qui intervient lorsqu'un client veut effectuer un appel distant vient du port utilisé par RMI. En effet, les pare-feux interdisent souvent l'accès à certains ports spécialisés, comme celui utilisé par défaut lors d'un appel RMI.

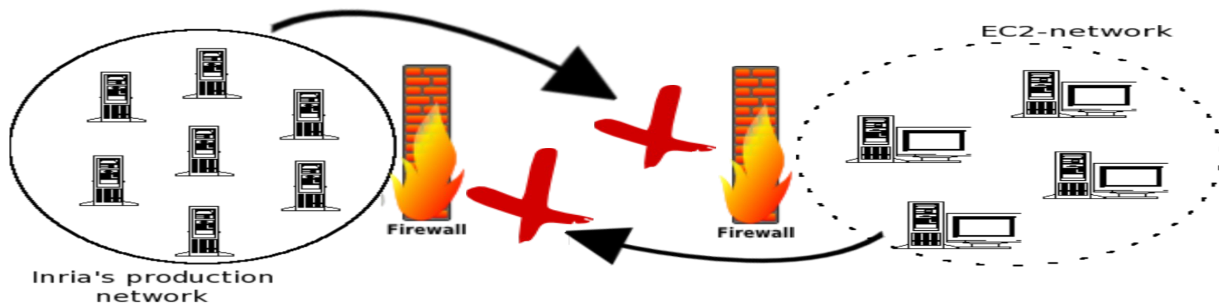
Les pare-feux sont capables de filtrer des paquets selon :

- Leur adresse source,
- Leur adresse de destination,
- Le port de destination,
- Le protocole couche transport (exemple : TCP, UDP) utilisé.

Donc on peut conclure que le protocole par défaut et sur lequel se base ProActive, RMI n'est pas adapté aux pare-feux pour la simple et importante raison que les appels RMI sur une machine distante se font sur un port aléatoire. Il y a donc une forte chance de trouver un port bloqué par des pare-feux interdisant la communication entre la machine locale et la machine distante.

Tout en essayant de déployer des noeuds sur EC2 on s'est heurté au problème de pare-feu déjà cité et ce fut le problème initial à résoudre pour atteindre notre but dans le cadre du travail d'études de recherche.

Dans notre cas d'utilisation les pare-feux interdisaient les communications entre le réseau de production de l'INRIA et le réseau de machines EC2.



VI.1) Étude des différentes solutions

Dans le but de résoudre le problème du pare-feu qui bloquait les communications aux machines distantes(EC2-virtual machine) nous avons élaboré et travaillé sur différentes solutions avec un regard critique sur chacune d'elles tout en détaillant leurs avantages et inconvénients.

VI.1.a) Tunnel SSH

Cette solution consiste à diriger les communications entre la machine locale et la machine distante via un tunnel SSH qui nous permet de contourner le problème de pare-feu tout en fixant le port avec lequel les machines distantes (EC2) et locales communiquent. Cette solution n'est pas adaptée à la programmation répartie vu le coté manuel de construction de tunnel SSH, et en plus on aura autant de tunnels SSH que de noeuds distants (EC2).

L'inconvénient majeur de cette solution est la limitation du nombre de noeuds distants. En effet, l'utilisation d'OpenSSH limite le nombre de tunnels SSH simultanés à 50 (donc, à raison de deux tunnels par noeuds (aller + retour), cela limite à 25 le nombre de noeuds EC2).

Le tunnel SSH qui reliait le réseau de production de l'INRIA et le réseau de EC2 se faisait en utilisant l'une des solutions suivantes :

Solution 1 : mise en place du tunnel SSH partant de la machine **crusoe** (machine appartenant au réseau de production de l'INRIA) vers une machine qu'on a déjà reservée sur EC2 :

```
ssh -i aparisy-keypair -L 1099:crusoe.inria.fr:1099 root@ec2-72-44-41-81.compute-1.amazonaws.com
```

Ce tunnel entre les deux réseaux n'a pas réussi à être créé pour la simple et bonne raison que le réseau de production de l'INRIA est inaccessible de l'extérieur pas SSH, et donc que les communications du réseau EC2 vers le réseau de l'INRIA ne traversent pas le pare-feu (la politique choisie par l'INRIA étant d'interdire les connections SSH depuis l'extérieur). C'est la où intervient le besoin d'indiquer au tunnel SSH (cette fois ci lancé depuis EC2 en *reverse*) la passerelle **ssh-sop** pour permettre la connection SSH vers l'INRIA depuis l'extérieur.

Solution 2 : tunnel en *reverse* de EC2 vers **crusoe** :

```
ssh -i aparisy-keypair -R 1099:crusoe.inria.fr:1099 root@ec2-72-44-41-81.compute-1.amazonaws.com ssh-sop.inria.fr
```

Afin de vérifier la création effective du tunnel et son bon fonctionnement on a utilisé cette commande pour voir la liste des connections SSH sur les ports de la machine EC2 et pour vérifier si les connections arrivant sur le port que l'on a précisé sont bien redirigées via le tunnel SSH :

```
netstat -lapte | grep LISTEN
```

Cette solution, bien qu'elle résolve le problème de pare-feu et augmente la sécurité dans l'envoi de messages grâce au protocole SSH qui comprend, dans ses principes, une couche de sécurité due au chiffrement et déchiffrement des communications passant par le tunnel, présente plusieurs inconvénients qui la rendent inutilisable dans notre propre cas d'utilisation.

Inconvénients :

- l'aspect manuel de la solution qui revient à créer manuellement les tunnels SSH pour faire passer les communications,
- Un surcoût dû au chiffrement et déchiffrement des messages,
- Un surcoût dû aux créations de tunnels SSH dans le cas où il y a un nombre important de noeuds externes (EC2),
- Limitation du nombre de noeuds distants due à la limitation du nombre de tunnels SSH possibles.

VI.1.b) rmissh

Ce protocole est basé sur RMI tout en créant des tunnels SSH pour contourner le problème de pare-feu et pour permettre les communications RMI entre machines locales et distantes.Cette solution est en elle même une automatisation de la solution du tunnel SSH avec des améliorations.

VI.1.b.1) Présentation

Le protocole rmissh permet aux utilisateurs de transférer l'ensemble de leurs communications RMI ou HTTP via SSH.

Ce type de fonctionnalité est utile pour deux raisons :

- il pourrait être nécessaire de crypter les communications RMI d'améliorer le modèle de sécurité RMI.
- la configuration du réseau, dans lequel une application est ProActive déployée, peut contenir des pare-feux qui rejettent directement les connexions TCP aux machines qui contiennent des objets actifs. Si ces machines sont autorisées à recevoir des connexions SSH sur leur port 22 (ou un autre numéro de port), il est possible de multiplexer et dé-multiplexer toutes les connexions RMI à cet hôte par l'intermédiaire de son port SSH.

Pour réussir à utiliser cette fonction avec des performances raisonnables, il est impératif de comprendre :

- la configuration du réseau sous-jacent: l'emplacement et la configuration du pare-feu.
- les modèles de communication des ProActive *runtime*
- la JVM dite source et d'où les communications RMI parviennent.
- la JVM dite destination qui reçoit les communications RMI.

Et c' est ici que le tunnel SSH intervient pour relier les deux JVM via le port SSH (22) pour faire parvenir les communications de la JVM source vers la JVM destination.

VI.1.b.2) Configuration du réseau

Étant donné deux réseaux A et B, pour faire du déploiement à distance, plusieurs questions se posent : A est il autorisé à ouvrir une connexion à B? B est il autorisé à ouvrir une connexion à l'A? (les réseaux sont rarement symétrique)

Si vous utilisez un protocole TCP ou UDP à base de protocole de communication (remarque : RMI est basé sur TCP), ces questions reviennent à demander : Quels ports de B nous permettent d'ouvrir une connexion à A et vice-versa...

Ce protocole a été élaboré pour relier deux réseau types A et B , A étant un réseau protégé par un pare-feu qui permet les connexions sortantes sans contrôle, mais ne permet que les connexions entrantes sur le port 22. Hôte B est également protégé par un pare-feu.

Cette solution est bien adaptée à notre cas d'application d'où les réseaux A et B représentent successivement le réseau de production de l'INRIA et le réseau de EC2 qui est accessible par le réseau de production via SSH comme vu précédemment.

Chaque fois qu'une demande est faite à un objet ProActive non-local , cette demande est effectuée avec le protocole de communication de la destination indiquée par la JVM. Notamment, chaque JVM est caractérisée par une unique propriété nommée **proactive.communication.protocol** qui est fixé à l'un des protocoles suivants:

- rmi
- http
- rmissh
- ibis
- Jini

Cette propriété permet d'identifier le protocole qui est utilisé par chaque client de la JVM pour envoyer des données à cette JVM. Pour utiliser différents protocoles pour les différentes JVM, deux solutions existent :

- modifier les descripteurs de déploiement ProActive et de transmettre les modifications à titre d'option de ligne de commande de la JVM :

```
<jvmProcess class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
....
<jvmParameters>
   <parameter value='-Dproactive.communication.protocol=rmissh'/>
</jvmParameters>
...
</jvmProcess>
```

- mettre dans le fichier de configuration ProActive sur l'hôte distant à la propriété **proactive.communication.protocol** le protocole désiré

```
<prop key='proactive.communication.protocol' value='rmissh'/>
```

VI.1.b.3) rmissh : protocole de communication.

Ce protocole est un peu spécial car il tient beaucoup de la compatibilité avec le protocole RMI et beaucoup d'options sont disponibles pour optimiser ce dernier.

Ce protocole peut être utilisé pour automatiquement transférer les communications RMI par le biais de tunnels SSH. Chaque fois qu'un client souhaite accéder à un serveur distant rmissh, plutôt que de contacter directement le serveur distant, il crée d'abord un tunnel SSH à partir d'un port aléatoire au niveau local vers le serveur présent sur l'hôte distant. Cette opération est nommée **port forwarding**. Il ne lui reste plus pour se connecter qu'à ce serveur est de se mettre en écoute sur le port local aléatoire choisi par le tunnel SSH. Le démon SSH qui tourne sur le serveur reçoit les données de ce tunnel et les transmet au serveur réel (local de son point de vue).

Ainsi, chaque fois que vous demandez une JVM qui est accessible uniquement par le biais de rmissh (à savoir, lorsque vous définissez son paramètre **proactive.communication.protocol** à rmissh), vous devez vous assurer que le démon SSH est en cours d'exécution sur son hôte.

Les propriétés les plus importantes pouvant être définies pour configurer le comportement du *tunneling* SSH sont listées ci-dessous. Toutes ces propriétés sont des propriétés du côté client :

- **proactive.ssh.port** : le numéro de port sur lequel le démon SSH attend les connections entrantes visant à accéder à la JVM. Si cette propriété n'est pas définie, la valeur par défaut est 22.

- **proactive.tunneling.try_normal_first** : si cette propriété est définie à **yes**, le code de *tunneling* essaie toujours de faire un lien direct RMI à l'objet distant avant d'utiliser le *tunneling* SSH. Si cette propriété n'est pas définie, la valeur par défaut est de ne pas faire ces tentatives de connexion directe. Cette propriété est particulièrement utile si vous souhaitez déployer un certain nombre d'objets sur un réseau local où seulement l'un des hôtes doit être lancé avec le protocole rmissh (afin de le rendre accessible hors du réseau local). Les autres hôtes situés sur le réseau local peuvent utiliser la propriété try_normal_first pour éviter d'utiliser le *tunneling* ce qui évite un surcoût à l'application.

Ce dernier paramètre est très important, en particulier en présence de pare-feux. Il permet en effet les communications entre 2 nœuds locaux en essayant tout d'abord le protocole RMI puis, en cas de non réception des messages après un certain timeout, en utilisant le *tunneling* SSH pour permettre les communications distantes.

Cette solution a le même surcoût que le *tunneling* SSH, bien que l'option **try_normal_first** permette de l'atténuer. En revanche, et comme expliqué précédemment, le réseau de production de l'INRIA n'est accessible de l'extérieur qu'à travers une passerelle (ssh-sop) et il n'est actuellement pas possible de spécifier une passerelle dans le protocole rmissh.

Au final, et compte-tenu de l'obligation de passer par une passerelle pour accéder de EC2 vers le réseau de production de l'INRIA et du fait que l'implémentation du protocole rmissh ne permette pas d'intégrer une passerelle lors du *tunneling* SSH entre machine locale et distante, cette solution n'est pas adaptée à notre cas d'utilisation.

VI.1.c) PAMR

VI.1.c.1) Vue d'ensemble de ProActive Message Routing

La programmation avec la plate-forme ProActive prend en charge plusieurs protocoles de communication tels que le RMI, HTTP, Ibis RMI ou SSH. Chaque protocole a ses propres avantages et inconvénients. Par exemple RMI est raisonnablement rapide, mais il n'est pas compatible avec les pare-feux. HTTP est plus lent, mais a seulement besoin d'un port TCP ouvert.

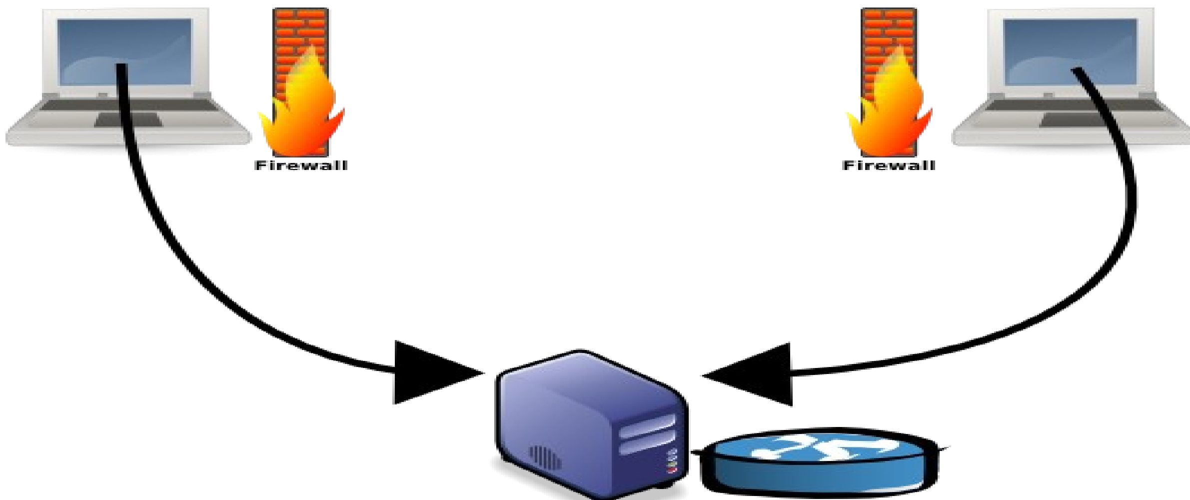
Le protocole PAMR a été développé par Jonathan Martin, puis maintenu et amélioré par Clément Mathieu. Il est conçu pour permettre le déploiement des applications ProActive seulement lorsque des connexions sortantes sont disponibles. De telles situations peuvent être rencontrées en raison de :

- pare-feux autorisant uniquement les connexions sortantes (ce qui est la configuration par défaut de nombreux pare-feux personnels),

- machines virtuelles avec une pile de réseau virtuel (EC2).

Lorsque le protocole de routage est activé, chaque ProActive *runtime* envoie ses communications au routeur qui joue un rôle central dans le transfert des communications. Ce routeur est maintenu actif, et la connexion au routeur est utilisée comme un tunnel pour distribuer les messages à leurs destinataires. Si le tunnel "tombe", il est automatiquement réouvert par le ProActive *runtime*.

Ce protocole est bien adapté à notre cas d'utilisation et nous permet de résoudre le problème de pare-feu tout en utilisant un routeur de messages dont l'unique rôle est de transférer les communications RMI.



VI.1.c.2) Configuration du ProActive Message Routing

La configuration du protocole ProActive Message Routing est un processus en deux phases. La première consiste à configurer et lancer le routeur de messages sur la machine intermédiaire faisant office de routeur. Ensuite, la deuxième étape est de configurer chacun des ProActive *runtime* afin qu'il utilise le protocole de ProActive message routing et le port sur lequel le routeur écoute et transfère les communications RMI.

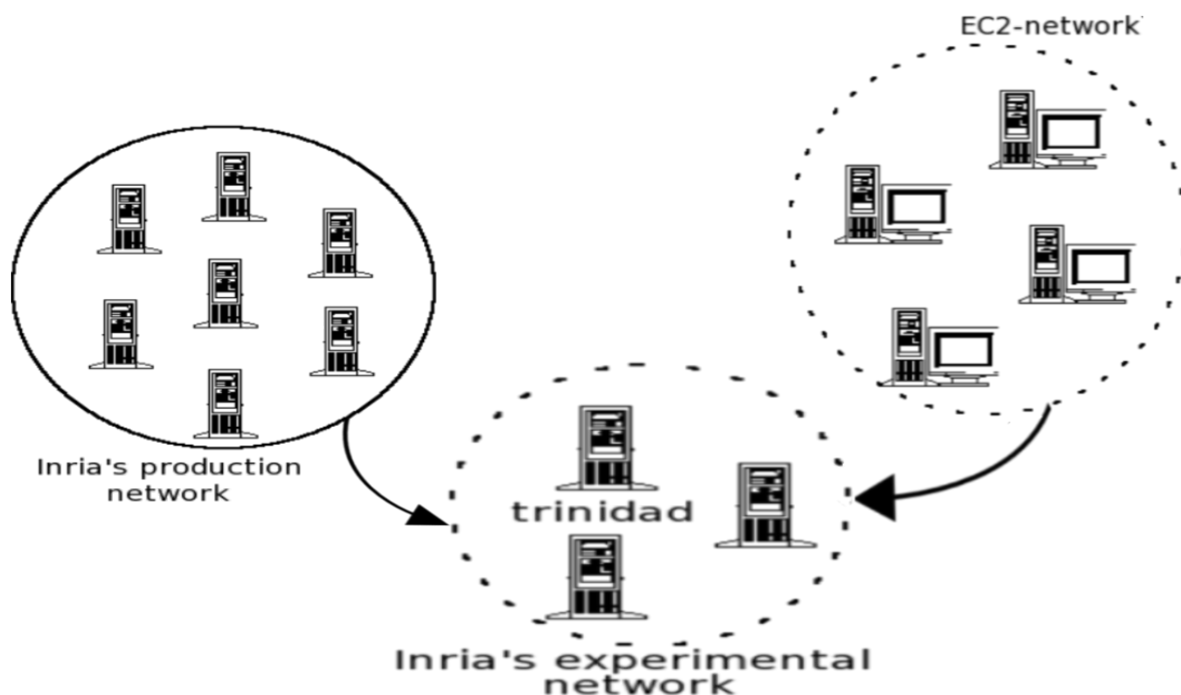
VI.1.c.3) Message Router Configuration :

Le routeur de messages doit être lancé sur une machine qui est accessible via TCP par tous les ProActive *runtimes* impliqués dans le calcul. La localisation du routeur de messages est donc essentielle. La machine doit être accessible (ports ouverts permettant les connexions TCP), avoir une bonne connexion au réseau et être stable. Si le message routeur plante, l'ensemble de l'application doit être redémarré.

VI.2) Solution retenue : PAMR

Dans notre cas d'utilisation, le routeur décrit précédemment doit être démarré sur une machine disposant de ports ouverts vers l'extérieur. Mais par souci de sécurité les entreprises créent souvent un réseau distinct ouvert vers l'extérieur et appelé réseau expérimental.
Ce réseau a la vocation d'être utilisé pour le genre d'applications nécessitant l'ouverture de ports permettant la communication avec le réseau extérieur. On peut rapprocher cette notion de celle de DMZ.

Pour la mise en œuvre de cette solution le routeur a été démarré sur une machine du réseau expérimental (trinidad). Le Scheduler est démarré sur le réseau de production, quelques nœuds locaux sont ajoutés au Scheduler, puis on démarre un nœud sur EC2 tout en indiquant dans le fichier de configuration de tous les nœuds locaux et distants (EC2) que le protocole utilisé est PAMR ainsi que l'adresse et le port sur lesquels ils vont communiquer avec le routeur.



La solution PAMR est bien adaptée pour notre cas d'utilisation bien qu'elle présente un défaut majeur qui est le fait que toutes les communications entre les differentes machines locales et distantes se font par le biais du routeur, ce qui est parfois inutile.

Donc dans le cas de noeuds fortements communicants cette solution peut s'avérer être trés coûteuse (doublement du nombre de communication). De plus, et dans le cas précis d'EC2, il y a un facteur économique due au fait que les communications avec les noeuds EC2 se payent.

VII) Perspectives

Grâce à PAMR, nous avons réussi l'intégration de nœuds provenant de réseaux hétérogènes au sein d'un unique RM. L'étape suivante est l'automatisation de ce processus. Cependant celle-ci est loin d'être triviale de par l'existence de plusieurs problèmes.

Tout d'abord d'ordre algorithmique : les stratégies permettant de lancer la réservation de nouvelles machines en cas de surcharge doivent prendre en compte le type d'instance requis (plus ou moins de CPU/RAM/bande passante par exemple) ce qui est déjà délicat a déterminer. De plus, il est difficile de prédire à l'avance le type de calculs restant à faire. Il n'est ainsi pas forcément optimal de lancer la réservation de machines pour une heure entière, par exemple, alors que le calcul risque de se terminer en quelques minutes.

Ensuite un autre problème plus "pratique" se pose : malgré la réactivité plutôt bonne des instances d'EC2, leur démarrage n'est évidemment pas instantané. Cette latence peut atteindre quelques minutes, ce qui bien que faible à l'échelle de calculs de plusieurs heures, peut s'avérer problématique et complique - d'autant plus l'automatisation (nécessité "d'anticiper" encore plus les calculs à venir). Et comme Brian nous l'a fait remarquer, ce travail d'automatisation pourrait faire l'objet d'une thèse à part entière.

Or, durant ce TER, nous avons fait le choix de nous concentrer sur les aspects pratiques afin d'obtenir, dans des délais courts, un résultat fonctionnel.

Mais malgré le travail accompli, il reste encore de nombreuses choses qu'il serait possible d'améliorer. En effet, et comme nous l'avons vu précédemment, PAMR n'est pas une solution viable dans tout les cas de figure, du fait en particulier d'un surcoût non négligeable dans le cas de nœuds fortement communicants. Il est donc nécessaire de trouver des solutions palliant à ce problème.

L'une d'entre elles pourrait être le déploiement de deux Schedulers (un sur chaque réseau interne) qui se communiqueraient les tâches en utilisant le protocole PAMR.

Mais la meilleure solution (en cours de développement par Arnaud Contes, principalement) serait que ProActive soit multi-protocoles. Dans le cas présent, l'idéal serait d'imiter le mécanisme de rmissh, mais en utilisant PAMR. Cela permettrait d'utiliser RMI lors des appels locaux (sans passer par le routeur), et PAMR lors des appels distants. Cette solution serait la plus approprié car elle n'entrainerait pas un surcoût important en "forçant" tout les échanges à passer par le routeur.

VIII) Remerciements

Nous tenions à remercier tout particulièrement nos deux encadrants, Messieurs Caromel Denis et Amedro Brian. Ils ont toujours été à notre écoute lors de nos différentes questions et nous ont aidé tout au long de ce TER

Nous remercions également toute l'équipe d'OASIS pour son accueil, sa bonne humeur générale qui a favorisé notre rapide intégration au sein de l'équipe et, de fait, aider à la réalisation de ce TER dans les meilleures conditions possibles.

Nous remercions également l'équipe d'ActiveEon pour son aide concernant la solution mise en oeuvre dans ce TER.

IX) Bibliographie

IX.1) Amazon EC2

Site officiel des *Web Services* d'Amazon :
http://aws.amazon.com/ec2/

Wiki détaillant les commandes basiques d'EC2 :
http://wiki.ipxwarzone.com/index.php5?title=Amazon_WebServices_EC2

Nous nous sommes également inspirés des travaux de Guillaume Laurent, un ancien ingénieur de l'équipe d'OASIS qui a travaillé sur EC2.

IX.2) PAMR

Chapitre du manuel de ProActive consacré au protocole PAMR :
http://proactive.inria.fr/release-doc/pa/multiple_html/pamr.html

De manière général, on à fortement utilisé le manuel de ProActive disponible à l'adresse suivante :
http://proactive.inria.fr/release-doc/pa/multiple_html/index.html

X) Annexes

X.I) Exemple de commandes de l'API EC2

<table>
<tr><th>AMI Tools</th><th>Image Tools</th><th>Instance Tools</th></tr>
<tr>
<td>ec2-bundle-image
Creates a bundled AMI from an operating system image created in a loopback file.
ec2-bundle-image -k private_key -c ec2_cert -u user_id -i image_path -r {i386 | x86_64} [-d destination] [-p ami_prefix] [--batch] [--kernel kernel-id] [--ramdisk ramdisk_id] [--block-device-mappingblock_device_mapping]</td>
<td>ec2-register
Registers the AMI specified in the manifest file and generate a new AMI ID.
ec2-register manifest</td>
<td>ec2-run-instances
Launches one or more instances of the specified AMI .
ec2-run-instances ami_id [-n instance_count] [-g group [-g group ...]] [-k keyname] [-d user_data | -f user_data_file] [--addressing addressing_type] [--instance-type instance_type] [--availability-zone zone] [--kernel kernel_id] [--ramdisk ramdisk_id] [--block-device-mapping mapping]</td>
</tr>
<tr>
<td>ec2-bundle-vol
Creates a bundled AMI by compressing, encrypting, and signing a snapshot of the local system's root file system.
ec2-bundle-vol -k private_key -u user_id -c ec2_cert -r {i386 | x86_64} [-s size] [-d destination] [-e exclude_directory_1,...] [-p ami_prefix] [-v volume] [--ec2cert ami_path] [--fstab fstab_path] [--generate-fstab] [--kernel kernel-id] [--ramdisk ramdisk_id] [--block-device-mapping mapping] [-b, --batch]</td>
<td>ec2-deregister
Deregisters the specified AMI. Once deregistered, the AMI cannot be used to launch new instances.
ec2-deregister ami_id</td>
<td>ec2-terminate-instances
Terminates the specified instances.
ec2-terminate-instances instance_id [instance_id ...]</td>
</tr>
<tr>
<td>ec2-upload-bundle
Uploads a bundled AMI to Amazon S3 storage.
ec2-upload-bundle -b s3_bucket -m manifest -a access_key_id -s secret_key [--acl acl] [--ec2cert certificate] [-d directory] [--part part] [--url url] [--retry] [--skipmanifest]</td>
<td>ec2-describe-images
Returns information about AMIs, AKIs, and ARIs. If no parameters are specified, information about all images is returned.
ec2-describe-images [ami_id ...] [-all] [-o owner ...] [-x user_id]</td>
<td>ec2-confirm-product-instance
Returns a boolean indicating whether the specified product code is attached to the specified instance. If it is attached, It returns true. Otherwise, it returns false.
ec2-confirm-product-instance product_code -i instance_id</td>
</tr>
<tr>
<td>ec2-download-bundle
Downloads the specified bundles from S3 storage.
ec2-download-bundle -b s3_bucket -m manifest -a access_key_id -s secret_key -k private_key [-p ami_prefix] [-d directory] [--url url]</td>
<td>ec2-migrate-image
Migrates an AMI from one region to another.
ec2-migrate-bundle -k private_key -c ec2_cert -o access_key_id -w secret_access_key --bucket source_s3_bucket --destination-bucket destination_s3_bucket --manifest manifest_path --location {US | EU} --ec2cert ec2_cert_path [--kernel kernel-id] [--ramdisk ramdisk_id] {--mapping-file mapping_file | --mapping-url mapping_url |--no-mapping} --region mapping_region_name</td>
<td>ec2-describe-instances
Describes the current state of the specified instance(s). If you do not specify instances, all your instances are included in the output.
ec2-describe-instances [instance_id|availability-zone ...]</td>
</tr>
</table>

X.II) Exemple d'instances

	Standard (applications les plus courantes)			High CPU (calcul intensif)	
	Petite	Moyenne	Grosse	Moyenne	Grosse
CPU (Core Unit)	1	4	8	5	20
RAM (Go)	1,7	7,5	15,0	1,7	7,0
Stockage (Go)	160	850	1690	350	1690
Plate-forme	32 bits	64 bits	64 bits	32 bits	64 bits
Performances E/S	Moyennes	Hautes	Hautes	Moyennes	Hautes
Prix ($ / heure)	0,10	0,40	0,80	0,20	0,80

Légende du tableau ci-dessus :

Chaque *EC2 Core Unit* correspond à un processeur d'une puissance équivalent à 1,0-1,2 Ghz.

Les Performances E/S (Entrées/Sorties) correspondent à la fois aux temps d'accès aux disque en lecture/écriture, mais également à la bande passante et la qualité de connexion des instances.

X.III) Script pour S3

```
#!/bin/sh


# Bundling image API key passed as argument
BUCKET="oasis-proactive-bucket"
IMAGE_NAME="proactive-image"


# test to determine if the keys are missing
if [ ! -f /mnt/pk-*.pem ]
then
	echo "Missing private key file - copy it in /mnt"
	exit 1
fi


# cleanup
rm -rf ~/proactive/lib.old > /dev/null 2>&1


# we make the bundle using the key
ec2-bundle-vol -d /mnt -k /mnt/pk-*.pem -c /mnt/cert-*.pem -u 083356613891 -r i386 -p
$IMAGE_NAME


# we upload that bundle using the correct ProActive manifest
ec2-upload-bundle -b $BUCKET -m /mnt/proactive-image.manifest.xml -a
0F326N2S93BJQBH1VDR2 -s $1


# we warn the user that an upload HAS TO BE followed by a register
echo; echo
echo "Don't forget to register the image : ec2-register $BUCKET/${IMAGE_NAME}.manifest.xml"
```

X.IV) Exemple de déploiement de calcul de matrice sur des noeuds ProActive :

```
public static void main(String[ ] args) throws ProActiveException {
	deployer = PAGCMDeployment.loadApplicationDescriptor(new File(args[0]));

	//Activate all virtual nodes
	deployer.startDeployment();

	//Wait for all the virtual nodes to become ready
	deployer.waitReady();
	GCMVirtualNode vn = deployer.getVirtualNode("matrixNode");
	int n = 1000, m = n / 2;

	ProActiveConfiguration.load();

	Matrix m0, m1, m2;
	Vector v0, v1, v2;

	// Creates and randomly fills in both matrix and vector
	m0 = new Matrix(n, n); m0.randomizeFillIn();
	v0 = new Vector(n); v0.randomizeFillIn();

	// Creates both submatrixs, with sizes m and n-m
	try {
		Object[] parameters1 = { m0.getBlock(0, 0, m, n - 1) };
		Node a = vn.getANode();
		System.out.println(a.getNodeInformation());

		m1 = (Matrix) org.objectweb.proactive.api.PAActiveObject.newActive(
			Matrix.class.getName(), parameters1,a);
		Node b = vn.getANode();
		System.out.println(b.getNodeInformation());
		Object[] parameters2 = { m0.getBlock(m + 1, 0, n - 1, n - 1) };
		m2 = (Matrix) org.objectweb.proactive.api.PAActiveObject.newActive(
			Matrix.class.getName(), parameters2,b);

		// Computes both products
		long begin = System.currentTimeMillis();
		v1 = m1.rightProduct(v0);
		v2 = m2.rightProduct(v0);
		// Creates result vector
		v1.concat(v2);
		long end = System.currentTimeMillis();

		System.out.println("Elapsed time = " + (end - begin) + " ms");
	} catch (Exception e) {
		e.printStackTrace();
	}
	System.exit(0);
}
```

X.V) Exemple de fichier de déploiement GCMD

```
<?xml version="1.0" encoding="UTF-8"?>
<GCMDeployment xmlns="urn:gcm:deployment:1.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:gcm:deployment:1.0
        http://proactive.inria.fr/schemas/gcm/1.0/ExtensionSchemas.xsd">

        <environment>
                <javaPropertyVariable name="user.home" />
        </environment>

        <resources>
                <group refid="sshLan">
                        <host refid="ComputeNode" />
                </group>
        </resources>

        <infrastructure>
                <hosts>
                        <host id="ComputeNode" os="unix">
                                <homeDirectory base="root" relpath="${user.home}" />
                        </host>
                </hosts>

                <groups>
                        <sshGroup id="sshLan" hostList="eon1 eon[17-20]" />
                </groups>
        </infrastructure>
</GCMDeployment>
```

X.VI) Exemple de fichier de déploiement GCMA

```
<?xml version="1.0" encoding="UTF-8"?>
<GCMApplication xmlns="urn:gcm:application:1.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:gcm:application:1.0
http://proactive.inria.fr/schemas/gcm/1.0/ApplicationDescriptorSchema.xsd">

        <environment>
                <javaPropertyVariable name="proactive.home" />
                <javaPropertyVariable name="user.dir" />

                <descriptorVariable name="hostCapacity" value="2"/>
                <descriptorVariable name="vmCapacity" value="2"/>
        </environment>


        <application>
                <proactive base="root" relpath="${proactive.home}">
                        <configuration>
                                <applicationClasspath>
                                        <pathElement base="proactive"
                                                relpath="dist/lib/ProActive_examples.jar"/>
                                        <pathElement base="proactive"
                                                relpath="dist/lib/ibis-1.4.jar"/>
                                        <pathElement base="proactive"
                                                relpath="dist/lib/ibis-connect-1.0.jar"/>
                                        <pathElement base="proactive"
                                                relpath="dist/lib/ibis-util-1.0.jar"/>
                                </applicationClasspath>
                        </configuration>

                        <virtualNode id="matrixNode" capacity="16"/>

                </proactive>
        </application>

        <resources>
                <nodeProvider id="test">
                        <file path="./GCMD.xml" />
                </nodeProvider>
        </resources>

</GCMApplication>
```