REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Université Aboubakr Belkaid-Tlemcen

Ministère de l'enseignement supérieur et de la recherche scientifique

Faculté des sciences de l'ingénieur

Département d'informatique

Mémoire pour l'obtention du titre d'ingénieur d'état en:

Informatique

CONCEPTION ET IMPLANTATION D'UNE GRILLE DE CALCULS TOLERANTE AUX PANNES AVEC PROACTIVE

Présenté par :

- Sebbane Ismaïl
- Benosman Réda

- *President :* Mr ABDERAHIM mohammed El Amine
- *Examinateur :* Mr BENADDA Belkacem
- *Encadreur :* Mr BENAMAR Abdelkarim

~ Année universitaire 2006 / 2007 ~

Dédi cace

A nos parents

A nos familles

A nos ami(e)s

Remerciements

Au terme de ce travail, il nous est agréable de présenter nos remerciements a tous ceux qui ont contribué d'une manière ou d'une autre a l'aboutissement de cette thèse.

Nous tenons d'abord a exprimé nos profonde remerciements à Mr l'encadreur A.Benammar enseignant à l'université de Tlemcen et notre encadreur de nous avoir proposé le sujet sur la grille de calcul avec ProActive.

Un grand merci pour Mr O.Sebbane enseignant à l'université de Tlemcen d'avoir lu intensivement le mémoire et aider a ça correction

Mr B.Benadda enseignant à l'université de Tlemcen d'avoir accepté d'examiner le travail avec beaucoup d'attentions et pour son aide .

A nos amis Mr H.Benmerzouka et Melle S.Abi-Ayad pour leurs aides, encouragements, et leurs sympathies.

A cet honorable Jury Mr M.Abderahim et Mr B.Benadda, Président et membres, nous manifesterons notre gratitude pour leur présence, et leur patience.

Enfin on ne pourra terminer sans remercier nos parents, qui sont pour beaucoup dans la réalisation de ce travail.

Table des matières

Table des matières

Table des matières

Table des matières

Liste des figures

Introduction générale

Dans le cadre de l'obtention du diplôme d'ingénieur en informatique, notre projet de fin d'études porte sur l'application d'une installation de **grille de calcul** (grid computing) en utilisant le middleware **ProActive** au domaine de calcul distribué et parallèle (Calcule de Pi, Formule d'antennes, etc).
Ce type de traitement ne semble pas très complexe sur une architecture classique, le projet se concentre sur la manière de mettre en jeu la technologie Grille en utilisant la bibliothèque ProActive pour résoudre le problème avec détection et réparation de panne.
Ce projet est consacré à une partie de recherche bibliographique servant à étudier le langage **Java,** la technologie Grille de calcul, la bibliothèque ProActive, et une autre partie consacrée aux architectures disponibles pour permettre de mettre en place une solution à la portée du client.
Le but pratique de notre travail est de créer une application distribuée de type n-tiers qu'on a appelé **UnivGrid** moyennant des postes clients, un serveur, et plusieurs autres machines interconnectées entre eux pour réaliser des calculs répartis et en parallèles.
Le mémoire que nous vous soumettons avec l'application UnivGrid qui l'accompagne est le fruit de plusieurs mois de recherche. Il est constitué de 5 chapitres :

- Le 1er chapitre explique un peu en détail la technologie « Grid Computing » (grille de calcul).
- Le 2eme chapitre est consacré à l'étude de ProActive, son installation, son fonctionnement et son utilisation.
- Le 3eme chapitre est réservé à l'architecture logicielle du projet où on explique en détails les différentes classes du programme.
- Le 4eme chapitre présente l'architecture logicielle du projet, qui explique en détail le fonctionnement de notre application.
- Le dernier chapitre montre comment se présente notre application UnivGrid, et les différent étapes de son lancement.
- A la fin du mémoire se présente deux annexes qui expliquent quelques technologies utilisées et un résumé sur le projet.

Chapitre 1 :
Grille de Calcul

1.1 Définition de Grille de Calcul :

Une grille de calcul (Grid computing) est un modèle de système d'information où les ressources sont totalement éclatées à travers un réseau et sont potentiellement qualifiées de [1] :

- **Partagées** : elles sont mises à la disposition des différents consommateurs de la grille et éventuellement pour différents usages applicatifs.
- **Distribuées** : elles sont situées dans des lieux géographiques différents.
- **Hétérogènes** : elles sont de toute nature, différant par exemple par le système d'exploitation ou le système de gestion des fichiers.
- **Coordonnées** : les ressources sont organisées, connectées et gérées en fonction de besoins (objectifs) et contraintes (environnements). Ces dispositions sont souvent assurées par un ou plusieurs ordonnanceurs.
- **Externalisées** : les ressources sont accessibles à la demande chez un fournisseur externe.

C'est une forme d'informatique distribuée, basée sur le partage dynamique des ressources entre des participants, des organisations et des entreprises dans le but d'exploiter la puissance de calcul (processeurs, mémoires, ...) de milliers d'ordinateurs afin de donner l'illusion d'un ordinateur virtuel très puissant (voir Fig. 1.1). Ce modèle permet de résoudre d'importants problèmes de calcul nécessitant des temps d'exécution très longs en environnement "classique".
Chacune des machines interconnectées est aussi appelée un nœud.

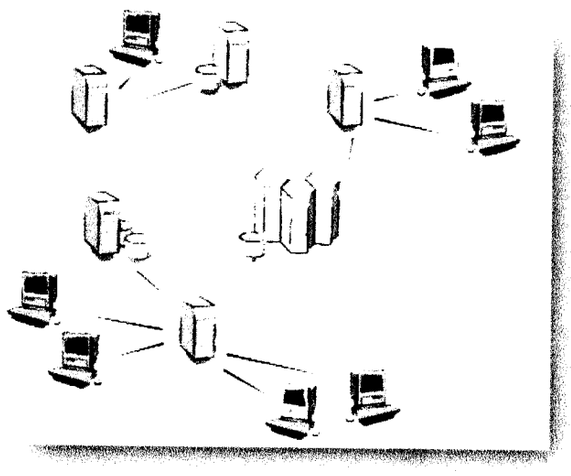

Fig. 1.1 grille de calcul []

1.2 Le But d'une grille de calcul :

Une grille de calcul est une infrastructure servant à délivrer des capacités de ressource (calcul, stockage et communication) de façon transparente à l'utilisateur, quand et où il les demande.
Ainsi on peut détailler plusieurs objectifs suivant les cas d'utilisations [2] :

- Grille de super ordinateurs ou de clusters interconnectés grâce à des liens de réseaux rapides. Ces derniers permettent le traitement rapide de calculs importants.

- Grilles de ressources inutilisées permettant d'exécuter un grand nombre de calculs indépendants, ceci de façon dynamique. Ce type de grille est typiquement utilisé sur Internet ou dans les réseaux d'entreprise pour utiliser au mieux les ressources inexploitées disponibles. Comme par exemple la nouvelle console de jeux PlayStation3 reliée à internet lance un client Folding@Home lorsqu'elle ne sera pas utilisée pour le jeu peut confier sa puissance de calcul à la recherche, c'est un projet de calculs distribués qui vise à étudier comment se « plient » les protéines. Les résultats de ces recherches devraient permettre de mieux comprendre certaines maladies comme le cancer, la maladie d'Alzheimer, et celle de la vache folle.

- Grilles collaboratives permettant la mise en commun de différentes ressources de divers types permettant une coordination immédiate des informations, celles-ci étant traitées séparément par ces antennes spécialisés.

1.3 Les différentes étapes d'un traitement de GRID :

Le lancement d'un traitement sur une grille se fait en plusieurs étapes : il faut d'abord sélectionner les nœuds qui participeront au calcul, puis les initialiser avec les données nécessaires (fichiers d'entrée, programmes, paramètres...), ensuite lancer les calculs, et enfin récupérer les résultats de ces calculs [2] (voir la figure 1.2).

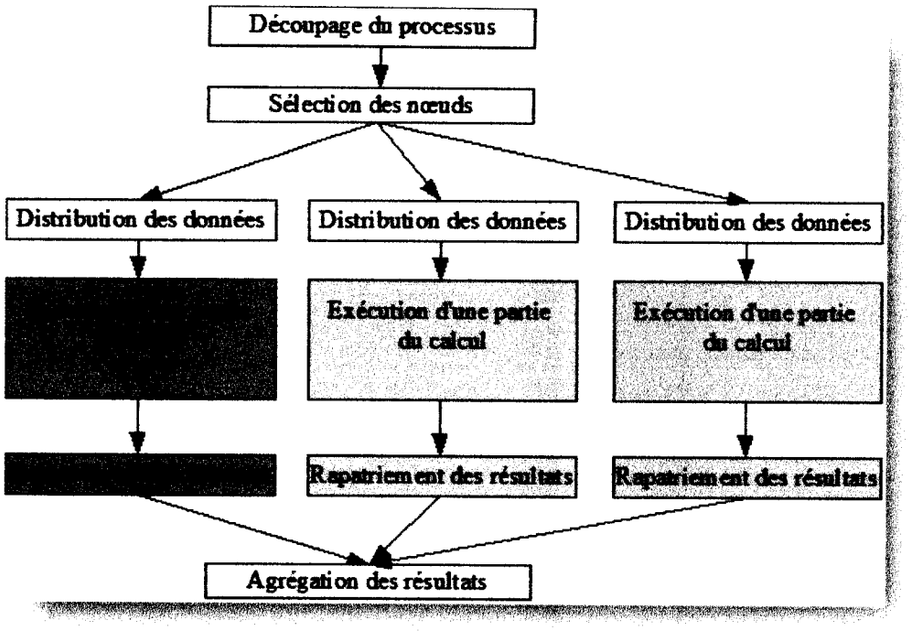

Fig. 1.2 traitement de la grille

1.3.1 Identification des nœuds :

Le système doit décider au moment où une requête lui parvient de choisir parmi les machines inscrites sur la grille celles qui vont participer au calcul, et cela peut se faire par le biais d'un fichier XML qui contient toutes les informations sur la grille de calcul. Un programme appelé ordonnanceur utilise ces informations pour choisir parmi les machines connectées celles qui vont effectuer les calculs demandés par le client.

1.3.2 Initialisation des différents nœuds :

Les nœuds sélectionnés par l'ordonnanceur doivent donc pouvoir lire les données qui les concernent avant d'effectuer les calculs. Il existe plusieurs techniques :

- On peut mettre en place un ou plusieurs serveurs de donnée où chaque nœud doit se connecter pour lire les données. L'inconvénient de cette technique c'est qu'on peut avoir la saturation du réseau au moment ou plusieurs nœuds tentent de se connecter au serveur.
- Une autre technique : chaque nœud copie les données qu'il doit traiter sur son propre espace de stockage local. Cette solution permet de limiter les accès réseau et le nombre de connexions simultanées au serveur de fichiers, qui sont souvent les éléments qui freinent le déroulement du calcul.

1.3.3 Exécution du traitement et récupération du résultat :

Sur chaque nœud choisi, on effectue des calculs, pour cela il existe différent type de traitement :
- On donne pour chaque nœud un même traitement et dès qu'un des nœuds fourni un résultat, soit en stoppe tout les autres et on fournit le résultat, soit on laisse terminer et on compare les différents résultats (cas de simulation)
- On donne pour chaque nœud une partie de calcul à effectuer, l'ordonnanceur se charge de regrouper les différents résultats pour fournir la réponse complète au client.
D'autre part, des mécanismes de surveillance des sorties d'erreur doivent être mis en place pour que l'ordonnanceur surveille et prenne les décisions nécessaires.

1.4 Les différents types de traitements :

Selon le type de traitement à appliquer aux données, la taille de ces dernières et leur quantité, plusieurs approches sont envisageables afin de décider les traitements que l'on va pouvoir paralléliser sur les différents nœuds de la grille.
La grille s'intéresse plus particulièrement aux aspects suivants[1,2] :

- Le stockage : il existe plusieurs manières pour conserver les données sur la grille :

  - Centralisée : sur un ou plusieurs serveurs, si les données ne sont pas très volumineuses
  - Distribuée : utilisée si un stockage centralisé pénaliserait les temps de lecture pendant les calculs, ou si le volume de donnée à stocker est trop important.

- L'utilisation des données: selon le type de stockage utilisé, l'accès aux données peut aussi être envisagé de différentes manières :

  - Si le stockage est centralisé, les accès peuvent êtres effectués directement sur l'espace de stockage, par exemple par la mise en place d'un VPN (Virtual Private Network).
  - Si le stockage est distribué, avant d'effectuer les calculs, les données sont copiées sur les différentes machines de la grille, ce qui permet de diminuer les accès réseau et accélérer les lectures pendant les calculs.

- Les calculs: de la même façon, on peut déterminer plusieurs cas de figures pour les calculs effectués sur les données à traiter.

  - Les calculs peuvent être effectués sur la première machine dont les ressources sont libres.
  - Les calculs peuvent être segmentés et lancés en parallèle sur plusieurs machines distantes, mais l'algorithme de calcul doit se prêter au découpage, (comme le cas de notre projet)
  - Lancer les même calculs sur les différents machines une ou plusieurs fois, pour à la fin comparer les résultats obtenus. Ce type de calcul assure une fiabilité maximale du résultat.

- Fournir le résultat : En fonction de la taille des calculs et du résultat, il est possible de fournir le résultat de manière :

  - Directe à l'usagé : Si la taille du résultat est raisonnable et le temps de calcul suffisamment réduit.
  - Indirecte : dans le cas contraire, où les calculs prennent plus de temps et fournissent un résultat dont la taille rendent plus problématique un rapatriement direct sur le poste de l'usager. Il faut envisager l'utilisation d'un serveur dédié au stockage provisoire de ces résultats que l'utilisateur. Après avertissement, on pourra récupérer ces résultats sans interférer avec les transferts de la grille liés aux autres calculs.

- La sécurité des transactions : il est nécessaire de protéger les données manipulées pour des raisons de sécurité et de confidentialité. D'une manière générale on doit aussi prendre en compte le fait que les opérations sur la grille peuvent faire intervenir un ensemble de commandes sur une machine quelconque susceptible de contenir d'autres informations sans rapport et tout aussi confidentielles.

On pourra baser la sécurité des transactions sur la grille sur plusieurs mécanismes à la fois :

  - Un système de certificats par utilisateur pour une transmission encryptée des données. Ce système implique un sous-système transparent de certificats provisoires pour les opérations de calcul sur la grille qui ne sont pas directement exécutées par l'utilisateur lui-même.
  - Une limitation des droits d'exécution sur les machines de la grille, même par un utilisateur autorisé, il n'est pas possible d'accéder à

certaines zones ou d'exécuter des commandes pouvant permettre d'accéder à des ressources locales sur le nœud de la grille.

De cette manière, on peut déjà assurer une bonne confidentialité des données et l'intégrité des nœuds de la grille. La façon dont on peut mettre ces mécanismes en place dépend ensuite fortement de l'implémentation de notre grille, ainsi que des protocoles mis en jeu.

1.5 **Exemple de fonctionnement de la grille** :

Le fonctionnement de la grille dépend de l'architecture choisie , la figure suivante nous montre un exemple :

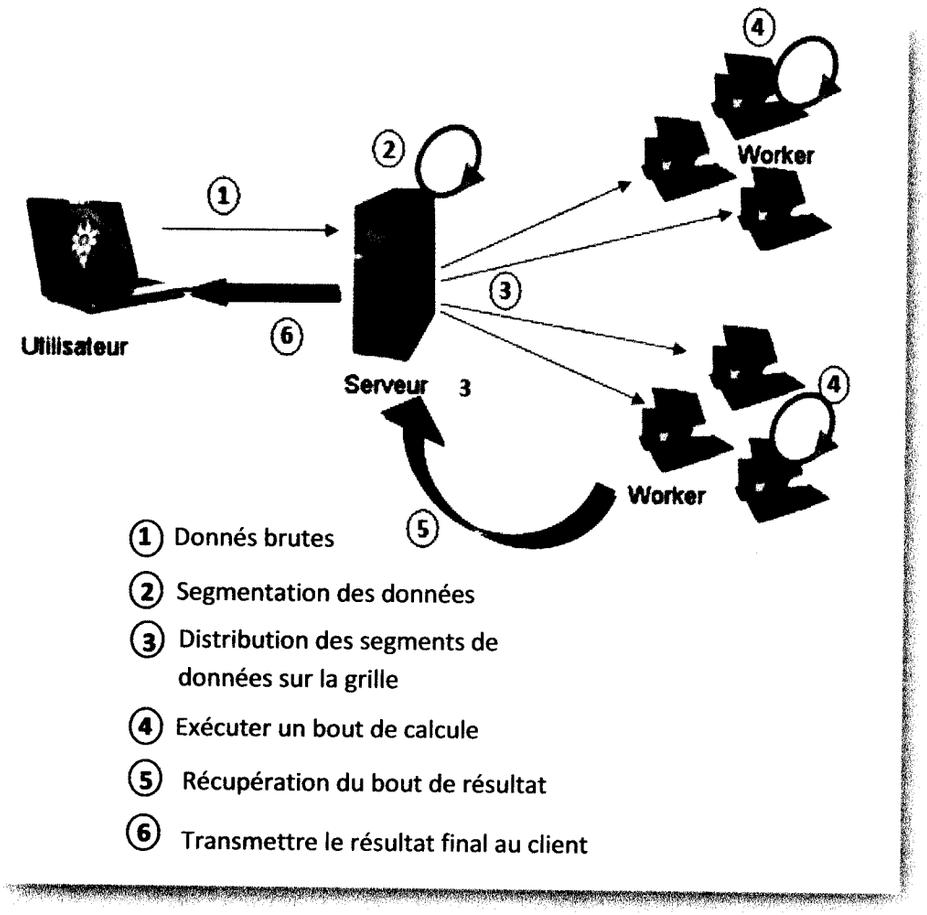

Fig. 1.3 Exemple de Fonctionnement.

1.6 Intégration et application au projet client

Le sujet du projet et les concepts de Grid Service ainsi que leur fonctionnement étant plus clairement définis, on se penche sur leur mise en place et leur application concrète.

1.6.1 Architecture fonctionnelle

L'application effectuant le traitement est installée sur un poste SERVEUR ce dernier va exploiter aux maximum toutes les ressources disponibles de notre grille ainsi chaque client n'a qu'à établir la connexion avec ce serveur pour effectuer ces traitements mais pour assurer tout ça les firewalls respectifs du serveur et des clients devront être adaptés pour que les communications puissent s'effectuer, et à la fin le client aura l'illusion d'utiliser une machine ultra puissante.

1.6.2 Les fonctions

On distingue dans l'application trois grands types de fonctions :

a) Les fonctions préliminaires aux calculs :

Ce sont des fonctions qui vont nous permettre de préparer les données d'entrée pour qu'elles puissent être traitées par notre programme de calcul .Cela consiste a récupérer les données à partir d'un endroit précis voire même les générer d'une manière aléatoire pour des raisons de test. Ces données vont être découpées en segments pour pouvoir les distribuer sur notre grille.
La méthode de segmentation des données d'entrée diffère car elle dépend fortement du type de données en entrées et de la fonction objective a résoudre. C'est la raison pour laquelle il est envisageable de prévoir plusieurs types de fonctions préliminaires.

b) Les fonctions de calcul :

C'est là le travail du programme principal installé sur chaque nœud, et c'est la puissance et la richesse en ressources qui va déterminer l'efficacité de ce dernier.

c) Les fonctions post-calculatoires :

Ce sont les fonctions responsables de la récupération des morceaux de résultats rendus par les calculs sur chaque nœud, et de leur réassemblage toujours en fonction du programme utilisé. Le résultat final sera renvoyé à l'utilisateur ayant fait la requête ou éventuellement stocké en attendant des traitements ultérieurs.

1.6.3 Les données :

On distingue quatre types de données:

a) Les données d'entrée fournies par l'utilisateur :

Pour qu'un client puisse transmettre ces données aux serveur le client bénéficiera d'une interface graphique une fois la connexion est établie avec le serveur. Après la validation du client ces données sont récupérées au niveau du serveur pour un traitement ultérieur.

b) Les données d'entrées des calculs intermédiaires :

c'est le résultat de l'application de la fonction de segmentation de données citée avant sur un flux de données envoyé par un client.

c) Les résultats des calculs intermédiaires :

Ce sont les résultats de chaque sous-calcul. Chacun de ces résultats est calculé sur un nœud particulier de notre grille.

d) Les résultats des calculs demandés par l'utilisateur :

Le serveur une fois qu'il reçoit les sous-résultats fournit par chaque nœud de la grille ,il va s'en charger de faire la réunion de ces derniers afin de générer un résultat final qui sera transmis par la suite au client ayant fait la requête.

1.6.4 Les flux

Les flux au sein d'une grille seront de trois types :

a) Flux d'initialisation des différents nœuds de la grille :

Il s'agit de la communication régulière entre chaque nœud de la grille et le serveur, qui permet à ce dernier de savoir à chaque moment l'état des machines qui forme la grille et ainsi on pourra faire une gestion dynamique qui va augmenter l'efficacité de notre système.

b) Flux de lancement des calculs sur la grille :

Ces flux, allant du serveur vers un nœud, sont à la fois les données d'entrée des calculs et des paramètres de calcul par un exemple le degré de précision.
Les données passées du serveur vers le client peu être sensible c'est la raison pour la quelle il faut envisager un mécanisme de transactions sécurisées pour plus de confidentialité.

c) Flux de transfert des résultats :

Ces flux seront le retour de traitement des données par chacun des nœuds de notre grille vers le serveur, ainsi que l'envoi du résultat final à l'utilisateur ayant fait la requête.
Les données transportées étant probablement sensibles, la sécurité de cette étape sera un point important.

1.6.4 Contraintes

Une première contrainte est la simplification de la méthode d'exécution sur les différentes machines. Ces dernières devant gérer elle-même les données à traiter, et ceci sans intervention humaine. Pour se faire, une première approche consistera en la mise en place d'un serveur de gestion des nœuds de la grille permettant de coordonner leurs actions.

Ce serveur sera également en charge du réassemblage du résultat final. Ce système de gestion de ressources de la grille conduira à une meilleure efficacité en fonction du type et de la charge de chaque machine et des données à traiter.

Le choix de la plateforme de traitement jouera un rôle primordial dans le cycle de vie de notre grille, ainsi on peut exploiter toutes les machines disponibles avec une totale transparence.

Le système de calcul distribué est en cours de développement par plusieurs groupes de recherche, celui-ci devra être extensible c'est la raison pour laquelle il faut prévoir une architecture de grille dynamique qui s'adaptera facilement à sont environnement .

Le temps est l'unité de mesure de performance de notre système de calcul un bon système est un système qui fournit de bons résultats dans un temps court ainsi des mécanismes d'optimisations seront intégrés a notre architecture.

Chapitre 2 :

Bibliothèque ProActive

2.1 Introduction :

La programmation d'applications à hautes performances nécessite la définition et la coordination de plusieurs activités parallèles. Une librairie pour la programmation parallèle se doit de fournir, non seulement une communication point à point, mais également des primitives de communication au sein de groupes d'activités. Dans le monde Java, RMI, le mécanisme standard de communication point à point, est approprié aux interactions de type client/serveur. Dans un contexte de calculs à hautes performances, des communications asynchrones et collectives doivent être accessibles au programmeur, ainsi le seul usage de RMI n'est pas suffisant.
ProActive est un environnement de développement sur grille se présentant sous la forme d'une bibliothèque 100% Java, permettant la programmation de calculs parallèles, distribués et concurrents. Il est aussi un middleware de grille [4].
Cette bibliothèque a été développée par l'INRIA de Sophia-Antipolis (la version 1 est sortie en 2001) dans le but de fournir une API complète simplifiant la programmation d'applications distribuées sur le réseau local, sur un cluster ou sur des grilles. ProActive est construit avec des classes Java standards et n'exige donc aucun changement de la machine virtuelle, il utilise cependant un *class loader* spécifique. Il utilise actuellement la bibliothèque Java RMI en tant que couche de transport par défaut. ProActive est open Source et s'inclut dans la communauté ObjectWeb [5]. On peut l'obtenir avec une documentation détaillée sur le site web suivant :
http://www-sop.inria.fr/oasis/ProActive.

2.2 Installation et utilisation de ProActive :

Pour pouvoir utiliser ProActive après l'avoir installé, il faut inclure tous les fichiers *jar* nécessaires à son fonctionnement dans la variable d'environnement *Classpath*. Lorsqu'on lance une application avec la machine virtuelle Java, il faut indiquer un fichier de politique de sécurité donnant les droits d'accès aux ressources du système. On précise aussi un fichier *log4j* qui configure la journalisation des erreurs de l'application, sinon un fichier par défaut est utilisé et
Voici un exemple de ligne de commande pour lancer un programme MonProgramme.java.

```
Java
-Djava.security.policy=$HOME/ProActive/scripts/proactive.java.policy
-Dlog4j.configuration=file:$HOME/ProActive/scripts/proactive-log4j
MonProgramme
```

2.3 Objet Actif :

Un objet actif est l'unité de base de distribution d'une application ProActive. Un objet actif a son propre fil de contrôle (activité) : une mémoire locale et un comportement spécifique qui gère les appels de méthodes, les stocke dans sa queue de requêtes et décide de l'ordre de leur service. Contrairement à Java standard, le programmeur en ProActive n'a pas à manipuler explicitement les threads pour gérer les échanges [6].

Un objet actif peut être créé sur n'importe quelle machine hôte. Une fois que celui-ci est créé, son activité et sa localisation (locale ou distante) sont complètement transparents, si bien qu'il est manipulé comme un objet passif standard.

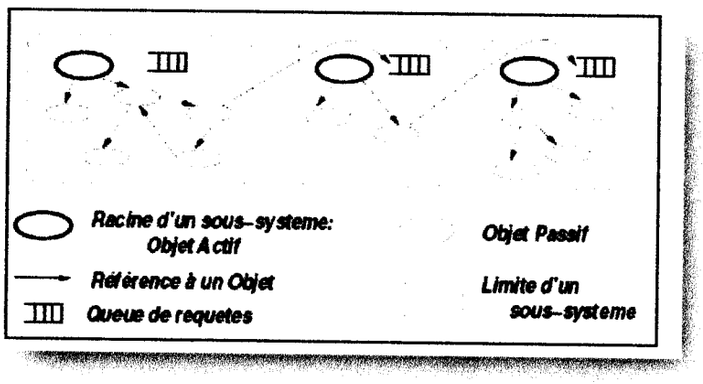

Fig. 2.1 Graphe d'objets avec objets actifs [7]

En ProActive une application distribuée est composée d'un certain nombre d'entités appelées objets actifs (figure 2.1). Elle est structurée en sous-systèmes (activités), pouvant être situés sur des machines différentes, coopérant les uns et les autres [7] :

- Chaque sous-système a un point d'entrée unique, sa racine qui est un objet actif ; les autres objets composant celui-ci sont des objets passifs. L'objet actif d'un sous système ne peut exécuter que ses propres méthodes (celles de l'objet actif lui-même et celles des objets passifs lui appartenant).
- Il n'y a pas de partage d'objets passifs entre sous-systèmes.

Ces caractéristiques ont des conséquences importantes sur la topologie des applications et sur la procédure de communication entre les sous-systèmes :

- De tous les objets composant un sous-système, il n'y a que l'objet actif qui est connu de l'extérieur de celui-ci.
- Tous les objets (actifs et passifs) peuvent avoir une référence sur un objet actif distant.
- Si un objet O1 a une référence sur un objet passif O2 alors O1 et O2 appartiennent au même sous-système.

C'est sur des objets appelés Nœud qu'on créer des objets Actifs, donc un Nœud et un objet java se situant dans une machine virtuel java et qui est référencée dans le registre RMI.
Un nœud possède donc une URL comme suit, accessible depuis n'importe quelle machine : **rmi://hote/nom_du_noeud.**
La création d'un Nœud en utilisant ProActive se fait comme suite :

```
Node node= NodeFactory.createNode("rmi://machine00/nom_du_noeud");
```

2.4 Anatomie d'un objet actif :

D'un point de vue de la structure interne, un objet actif est composé de plusieurs méta-objets (voir la figure si dessous).

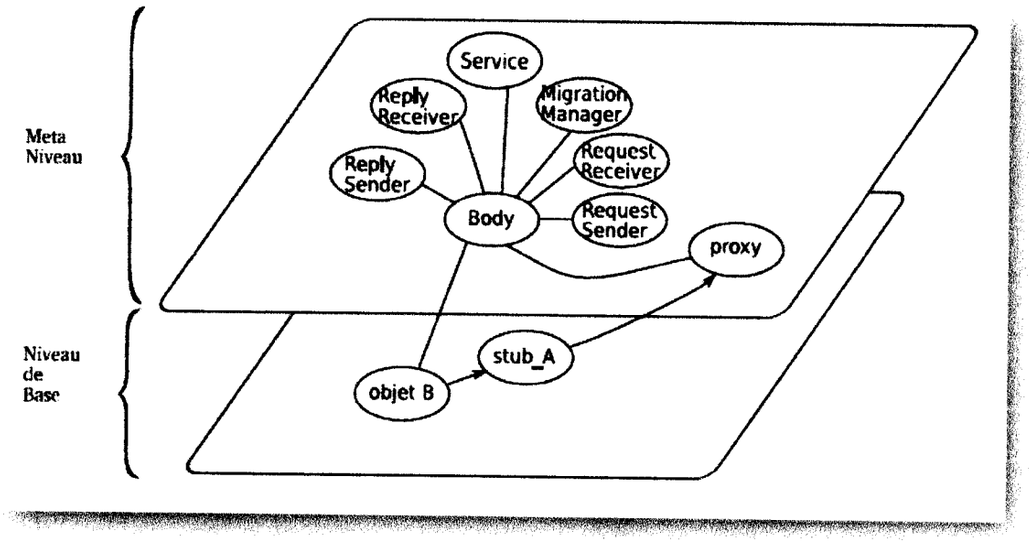

Fig. 2.2 Anatomie d'un objet actif [8]

À tout objet actif sont associés un Proxy et un Body. Le Body est le point d'entrée d'un objet actif. C'est la seule partie visible et accessible d'un objet actif. Le body sert de chef d'orchestre et gère le comportement de tous les autres méta-objets. L'ensemble Stub-Proxy est ce qu'on peut appeler une référence vers un objet actif. Il permet de masquer la notion de référence locale ou distante. Il sert à relayer les communications provenant d'un objet (actif ou non) vers un objet actif. Le stub est un objet java qui est polymorphiquement compatible avec l'objet rendu actif. Ce polymorphisme est assuré par héritage. Le stub est créé dynamiquement lors de la création d'un objet actif. Il est la représentation locale d'un objet distant.

La figure si dessus ***(Fig. 2.2)*** présente les divers objets et méta-objets traversés lors d'un appel de méthode d'un objet actif sur un autre. Premièrement, le stub intercepte les appels de méthode vers l'objet actif qu'il représente et réifie l'appel de méthode (1). Réifier un appel de méthode consiste à transformer un élément abstrait (l'appel de méthode) en un objet concret afin de pouvoir le manipuler. Une fois l'appel de méthode réifié, il est envoyé au proxy (2) qui contient tout le code nécessaire pour transférer l'appel de méthode au body de l'objet actif cible. Notamment, le Proxy crée un objet de type Requête qui va encapsuler l'appel de méthode ainsi que d'autres paramètres qui permettent, entre autre chose, de gérer la transmission du résultat de l'appel de méthode s'il y en a un ou encore l'asynchronisme de cet appel. Le Proxy passe la Requête au body de l'objet appelant (2) qui le transmettra à son RequestSender (3). Le RequestSender est le méta-objet en charge

de l'expédition de la requête au body. La requête est envoyée au body de l'objet actif distant (4). Quand un body reçoit une requête, il la passe à son RequestReceiver (5). Le RequestReceiver place ensuite la requête (6) dans la queue des requêtes. Le méta-objet Service représente l'activité de l'objet actif. Il contient la thread Java et sert les requêtes en les récupérant dans la queue des requêtes suivant le comportement qui lui a été défini. Si l'exécution d'une méthode renvoie une valeur de retour, il crée la réponse et la passe au méta-objet chargé de l'expédition des valeurs de retour. Symétriquement au RequestSender et au RequestReceiver, il existe des méta-objets qui gèrent l'envoi et la réception des réponses : le ReplySender et le ReplyReceiver [8].

2.5 Concept d'objet Futur

Lorsqu'un programme fait un appel de méthode sur un objet actif renvoyant un objet en retour, l'application n'est pas bloquée jusqu'à l'arrivée de la réponse, car un objet *futur* est envoyé tout de suite, et sera remplacé par l'objet attendu lorsqu'il sera disponible. Si le programme tente d'accéder à la valeur de l'objet *futur* (avant que sa vraie valeur ne soit connue), il sera bloqué. Ce mécanisme, appelé *wait-by-necessity*, permet au programme de se poursuivre jusqu'au moment où il aura réellement besoin du résultat, c'est une communication asynchrone entre objets. Le résultat doit être sérialisable pour traverser le réseau. De plus, ce mécanisme ne peut fonctionner que si l'objet résultat est *réifiable* (au sens de ProActive)[7,9]. C'est-à-dire s'il respecte les trois points suivants :

- Il n'est pas de type primitif (*boolean, int, float* ...),
- La classe de l'objet n'a pas l'attribut *final* et ne possède pas de méthode de type *final*
- Il a une méthode *constructeur* vide et sans argument.

Si on souhaite retourner un résultat de type primitif, il faut l'encapsuler dans un objet *réifiable*

Par exemple on utilise **FuturString** comme objet de retour et non **String** car ce dernier n'est pas

réifiable (la classe **String** possède l'attribut *final*), donc on ne pourrait pas en faire un objet ***futur***

```
//classe permettant la création d'objets réifiables de type String
public class FuturString implements Serializable{
      private String chaine;
      public FuturString() {}  //constructeur par défaut et sans argument
      public FuturString (String chaine){    //autre constructeur parameter
            this.chaine=chaine;
      }
      public String toString(){
            return(chaine);}}
```

2.6 Scénario de communication :

-Lorsque l'objet B est actif, A possède une référence vers un Stub de type B, et lui transmet l'appel de méthode.
-Le Stub transforme l'appel de méthode en un objet, et l'envoie au Proxy.
-Le proxy transforme l'appel de méthode au Body, et s'assure que l'appel est placé dans la file.
Puis, il teste si la méthode est ***void*** au si au contraire, elle renvoie un résultat, et le cas échéant, crée un futur du bon type.
-Sur le nœud de l'objet actif B, le Body est programmé pour placer les appels les uns après les autres en queue de la file (FIFO par défaut mais peut très bien être reprogrammé).
-Enfin, le véritable objet B exécute un par un les appels de la file sur son propre Thread, et, pour toutes les fonctions non ***void***, place le résultat dans le futur [10] (voir Fig. 2.3).

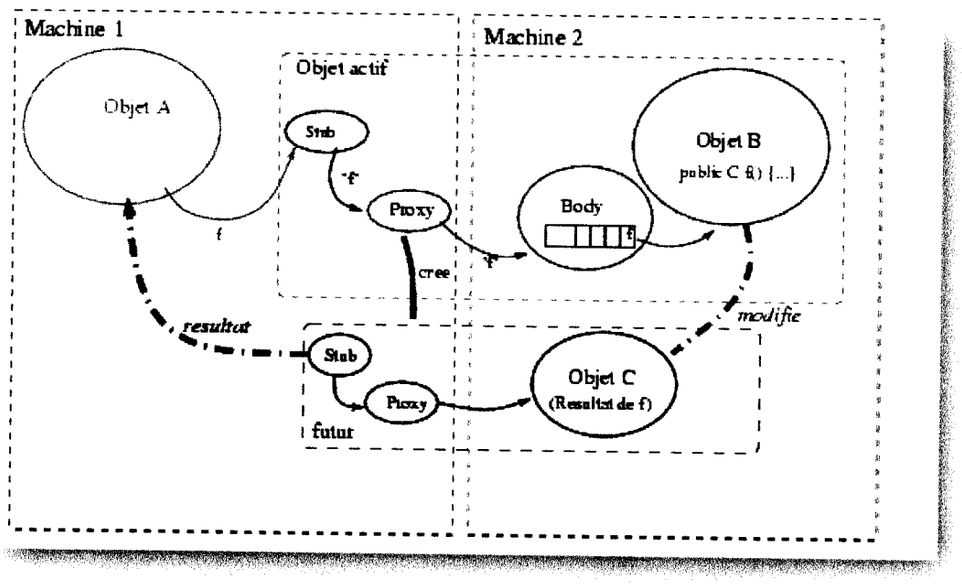

Fig. 2.3 Scénario de communication [10].

2.7 Création d'un objet actif :

L'instanciation d'un objet java standard se fait par l'instruction :

```
A  a = new A (2, "salut" );
```

Si nous voulons lui ajouter les fonctionnalités que proposent les objets actifs, il suffit seulement de modifier le code responsable de l'instanciation de cet objet. Il existe trois manières d'instancier un objet actif à la place d'un objet java standard :

1. La méthode la plus simple pour créer un objet actif est la suivante :

```
Object[] params = new Object[] {new Integer(26), "essai"} ;
Try {
A a = (A) ProActive.newActive(A.class.getName(), params);
} catch (Exception e) {e.printStackTrace();}
```

2. L'approche Instanciation-based permet d'utiliser une classe java standard et d'instancier un objet actif sans avoir à modifier ni le code, ni le bytecode de la classe.

```
Object[] params = { new Integer(2), "salut"};
A a = (A) ProActive.newActive("A", params, node);
```

3. L'approche Class-based est la plus statique. Elle implique la création d'une nouvelle classe qui implémente l'interface marqueur Active et hérite le cas échéant d'une classe préalablement existante. Active étant une interface marqueur pure, elle n'impose aucune contrainte au développeur. Il existe cependant des interfaces (EndActive, InitActive, RunActive étendant l'interface Active. Ces interfaces comportent des méthodes permettant de modifier le comportement par défaut des méta-objets de l'objet actif.

```
public class PA extends A implements Active {...}
Object[] params = { new Integer(2), "salut"};
A a = (A) ProActive.newActive("PA", params, node);
```

Le code d'instanciation d'un objet actif selon l'approche Class-based est le suivant :
Le premier paramètre est le nom qualifié de la classe (nom du package + nom de la classe). Le deuxième paramètre est un tableau d'objets qui contient les paramètres à passer au constructeur lors de la création de l'objet actif. Le troisième paramètre spécifie le noeud sur lequel l'objet actif va être créé. Ce noeud peut être local ou distant.

4. L'approche Object-based permet de rendre actif un objet java déjà instancié. Il est ainsi possible de rendre actif un objet java dont le source n'est pas disponible.

Cette approche réalise une copie profonde de l'objet passé en paramètre qui continuera d'exister au sein de la machine virtuelle. Si des objets avaient des références sur l'objet avant son activation, les références pointeront toujours sur l'objet initial et non sur la copie rendue active.

```
A a = new A (2, "salut" );
a = (A) ProActive.turnActive(a, node);
```

2.8 Enregistrement des objets actifs :

Un objet actif créé peut être utilisé par un autre objet se trouvant sur un autre noeud. Mais pour cela il faut qu'il soit *enregistré* grâce à la méthode **register()** de ProActive. Cette opération d'enregistrement ne peut se faire que localement sur le noeud où est créé l'objet. Par exemple, un objet **Obj_a** est créé localement sur machine00 :

```
A Obj_a = (A) ProActive.newActive(A.class.getName(), params);
```

Ensuite il est enregistré sur la machine00 avec l'identifiant "**TheObj_a**" :

```
ProActive.register(Obj_a, "//localhost/TheObj_a");
```

L'objet distant utilise alors la méthode **lookupActive()** pour rechercher l'objet **Obj_a** identifié "**TheObj_a**" et obtenir une référence sur celui-ci.

Par exemple, on exécute sur la machine machine01:

```
A Obj_a1 = (A)
ProActive.lookupActive(A.class.getName(),"//machine00/TheObj_a");
```

...[9]

2.9 Exemple d'application :

Voici un exemple Client/Serveur qui résume les notions vues précédemment :
Cet exemple contient 2 classes, la classe Hello qui s'exécute sur le post serveur et qui se charge de créer un objet Active et de l'enregistrer afin qu'elle puisse êtres accessible a partir de la classe ***HelloClient*** qui s'exécute sur un autre poste client [11].

2.9.1 Le Serveur :

```
import org.objectweb.proactive.*;
umport java.text.*;

public class Hello
{
private String name;
private String hi = "Hello world";
private DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy
HH:mm:ss");

public Hello() {} // constructeur par défaut et sans parameters

public Hello(String name) { this.name = name; }
public String sayHello() {
return hi + " at " + dateFormat.format(new java.util.Date())+
" from node : " + ProActive.getBodyOnThis().getNodeURL();
}  // constructeur qui retourne la date et l'URL de l'objet

public static void main(String[] args) {
try {
// creation d'un objet Active local
Hello hello = (Hello)ProActive.newActive(Hello.class.getName(),
new Object[]{"remote"});
java.net.InetAddress localhost = java.net.InetAddress.getLocalHost();
ProActive.register(hello, "//" + localhost.getHostName() + "/Hello");
// Enregistrement de l Objet Active sur le poste serveur
}
catch (Exception e) {e.printStackTrace();}
}
}
```

2.9.2 Le Client :

```
import org.objectweb.proactive.*;
public class HelloClient
{
public static void main(String[] args) {
Hello myServer;
String message;
try {
//  recuperation de l'adresse du serveur
if (args.length == 0) {
// l'argument est vide donc l'objet va être créer localement
myServer =
(Hello)org.objectweb.proactive.ProActive.newActive(Hello.class.getNam
e(),
new Object[]{"local"});
} else {
// rechercher de l obet Hello sur le serveur
System.out.println("Using server located on " + args[0]);
myServer =
(Hello)org.objectweb.proactive.ProActive.lookupActive(Hello.class.get
Name(), args[0]);
}
// invoquation de la méthode distante
message = myServer.sayHello();
// écrire le message
System.out.println("The message is : " + message);
}
catch (Exception e) { e.printStackTrace();}
}
}
```

2.10 Migration des activités

La mobilité d'une application est un paradigme de programmation qui consiste à donner la possibilité aux divers éléments d'une application de changer de localisation en cours d'exécution.
La migration permet le déplacement des objets actifs d'un noeud vers un autre. La migration telle qu'elle a été implantée au sein de ProActive est dite faible ; le code est mobile mais pas le contexte d'exécution.
Le concept de migration faible s'oppose au concept de migration forte.
La migration dite forte permet la migration d'un objet à n'importe quel moment pendant l'exécution de l'objet. Cette restriction est due au langage Java qui ne permet pas la capture des contextes d'exécution des Threads. La conséquence immédiate est l'impossibilité de faire migrer un objet tant que ce dernier n'a pas atteint un état désigné comme stable.
N'importe quel objet actif a la possibilité de migrer ou d'être migré.
Si l'objet actif possède un graphe d'objets passifs, ces derniers seront déplacés également vers la nouvelle localisation [12].
Le processus de migration repose actuellement sur la sérialisation Java pour transférer les objets d'une localisation vers une autre. C'est pour cela que tous les objets (actifs et passifs) doivent implanter l'interface marqueur ***Serializable*** afin que le processus de sérialisation puisse les transférer.
Le processus de migration peut être déclenché soit par l'objet actif lui-même, soit par un objet extérieur à l'objet actif. Il existe une API permettant de programmer la migration d'une activité.
Cette API est constituée d'une seule méthode nommée ***migrateTo*** ; méthode statique de la classe
ProActive. Afin de faciliter l'utilisation de la migration, la méthode ***migrateTo*** est surchargée pour former deux ensembles de méthodes statiques.

Le premier ensemble réunit les méthodes qui seront appelées à l'intérieur de l'objet actif. On utilise pour cet ensemble les méthodes suivantes :

```
// Migration vers le noeud designe par l'URL nodeURL
migrateTo(String nodeURL)
// Migration vers le noeud node
migrateTo(Node node)
// Migration vers le noeud qui heberge l'objet actif activeObject
migrateTo(Object activeObject)
```

Le second ensemble de méthodes réunit les méthodes qui seront appelées de l'extérieur de l'objet actif. Dans ce cas, l'objet appelant doit posséder une référence sur le Body de l'objet actif qu'il veut faire migrer.

```
// Migration de bodyToMigrate vers le noeud node
migrateTo(Body bodyToMigrate, Node node, boolean priority)
// Migration de bodyToMigrate vers le noeud node
migrateTo(Body bodyToMigrate, String nodeURL, boolean priority)
// Migration de bodyToMigrate vers le noeud qui heberge l'objet
//actif activeObject
migrateTo(Body bodyToMigrate, Object activeObject, boolean priority)
```

Le paramètre *priority* permet de modifier le comportement de prise en compte par l'objet actif du processus de migration. Si le paramètre est positionné à vrai, la requête de migration sera prioritaire sur toutes les requêtes déjà mises en attente dans la queue des requêtes de l'objet actif. Si le paramètre est positionné à faux, la requête sera traitée en priorité normale, c'est à dire qu'elle sera ajoutée à la suite des requêtes déjà en attente au sein de l'objet actif.

La migration d'une activité pose le problème de connectivité entre les objets. Le processus de migration doit assurer que la migration d'un objet actif n'entraînera pas des pertes de connectivité.

Si un objet possède une référence sur un objet actif, la migration de ce dernier doit être faite de manière à ce que l'objet qui n'a pas migré puisse continuer de communiquer avec l'objet qui a migré. Le deuxième problème est celui de la localisation d'un objet actif. On doit pouvoir retrouver un objet qui a migré et cela même si, initialement, on ne possédait pas de référence sur celui-ci. Pour palier ces problèmes, deux solutions ont été introduites au sein de la bibliothèque :

(1) les répéteurs et le serveur de localisation. Un répéteur est une référence laissée par l'objet actif au moment de la migration. Ce répéteur pointe vers le nouvel emplacement de l'objet actif.

Il reçoit les messages à la place de l'objet qui a migré et les renvoie à ce dernier. Ce mécanisme est mis en évidence par la ***figure 2.3***. Si l'objet migre plusieurs fois, on assiste à la création d'une chaîne de répéteurs. Plus la chaîne est longue, plus la probabilité qu'un des maillons de la chaîne disparaisse suite à des problèmes réseaux ou des problèmes sur la machine contenant le maillon augmente. Les longues chaînes entraînent une baisse des performances à cause des nombres de sauts à effectuer. La formation de chaînes de répéteurs est donc à éviter. La méthode utilisée par ProActive s'appelle le ***tensioning*** et

consiste lors du premier appel de méthode après migration à remplacer le lien vers le premier élément de la chaîne des répéteurs par un lien direct vers le nouvel emplacement de l'objet actif [8].

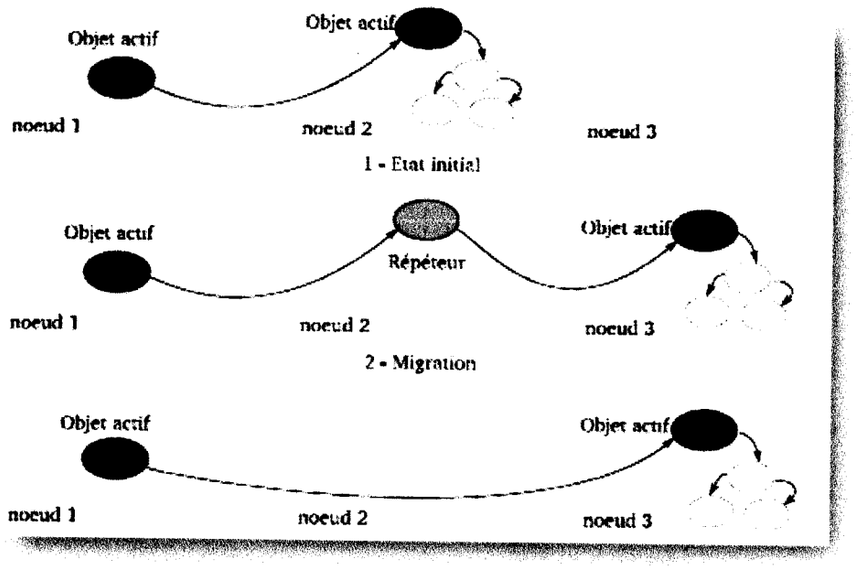

Fig. 2.3 Tensioning[8].

Le second mécanisme se base sur un serveur de localisation ***(figure 2.4)*** qui possède une référence sur chaque objet actif présent dans le système. A chaque migration, l'objet actif envoie sa nouvelle localisation au serveur. Après la migration d'un objet actif, les références que pouvaient posséder d'autres objets sur celui-ci deviennent invalides. Lors du premier appel de méthode sur un objet qui a migré l'appel va échouer. A ce moment, le mécanisme de localisation va de manière transparente :
(1) contacter le serveur de localisation afin de connaître le nouvel emplacement de l'objet actif qui a migré.
(2) mettre à jour la référence qu'il possédait.
(3) effectuer de nouveau l'appel de méthode. Contrairement à l'approche par répéteurs, on introduit des messages supplémentaires envoyés par l'objet actif qui a migré et par le mécanisme de localisation lors de l'échec d'une communication.

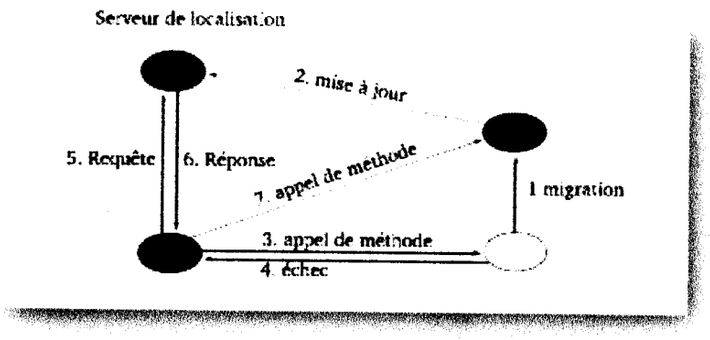

Fig. 2.4 Migration avec serveur de localisation[8].

2.11 Activités, contrôle explicite et abstractions :

Par défaut un objet rendu actif devient un serveur qui se contente de répondre aux requêtes dans l'ordre ou elles arrivent (FIFO).
Il est possible d'ajouter un comportement a un objet et ainsi de le rendre actif (client-serveur) en implémentant l'une ou toutes les interfaces :
1-InitActive
2-RunActive
3-EndActive

2.11.1 Exemple :

```
import org.objectweb.proactive.*;
public class A implements InitActive, RunActive {
private String myName;
public String getName() { return myName; }
public void initActivity(Body body) { myName = body.getName(); }
public void runActivity(Body body) {
int n = 0;
Service service = new Service(Body);
while (body.isActive()) {
n = service.getRequestCount();
// n est le nombre de requêtes présent dans la file
service.serveOldest() ;
service.waitForNewRequest ();
 // Ici: code de l'activite de l'objet...
}
} //..
```

```
public static void main(String[] args) throws Exception {
A a = (A) ProActive.newActive("A",null);
System.out.println("Name = "+a.getName());
}
```

Les appels distants étant asynchrones, ils sont mémorisés sous la forme de *requêtes* dans une file d'attente du côté de l'appelé. Ultérieurement, nous dirons que la requête est *servie* par l'objet actif. Jusqu'à présent, nous avons créé un objet actif sans préciser son activité, et par défaut un service fifo des requêtes était alors réalisé : exécution des requêtes dans l'ordre d'arrivée. Dans ce sens, les objets actifs sont bien des processus purement séquentiels.

La création d'objets actifs de type **Instanciation-based (2.7.2)** permet de donner à un objet un comportement spécifique : activité propre (qui ne consiste pas à servir des méthodes publiques), activité mixte (services et activité propre), services purs mais de type non-fifo.
L'exemple 4.4 présente un simple tampon borné. La routine ***runActivity*** permet à l'utilisateur de spécifier, sous la forme d'un thread explicite, l'activité

oid serveOldest ()	Sert la plus ancienne requête, bloqué
void serveOldest (String S)	Sert la plus ancienne de nom S, bloquée
void serveOldest (String S1, String S2)	Sert la plus ancienne de nom **S1,S2**
void serveYoungest ()	Sert la plus récente requête, bloqué
void serveYoungest (String S)	Sert la plus récente de nom S, bloquée
void serveYoungest (String S1, String S2)	Sert la plus récente de nom **S1,S2**
void serveOldestTimed (int t)	Service bloquant pour au plus **t** ms
void waitForNewRequest ()	Attente non active d'une requête

La routine ***runActivity*** permet à l'utilisateur de spécifier, sous la forme d'un thread explicite, l'activité de l'objet. Lorsque cette routine existe, elle remplace totalement le comportement fifo fourni par défaut. Le paramètre service de l'exemple précédent donne accès à tout un ensemble de routines de services permettant de sélectionner la requête à servir [13].

La méthode ***serveOldest*** est utilisée, mais toute une bibliothèque est accessible au programmeur. On y trouve des services *bloquants*, *non-bloquants*, *temporisés*, etc. mais également des ***itérateurs*** sur la file d'attente permettant d'étendre la bibliothèque si nécessaire. Notons ***waitForNewRequest*** qui permet de programmer une attente non-active.

Nous sommes ici en présence d'une programmation de l'activité qui est explicite,avec service lui aussi explicite, sans non-déterminisme sur l'ordre des services, dans l'exemple précédent on impose de servir la requête la plus ancienne. Ce type de programmation est fort utile si l'on souhaite avoir un contrôle fin sur l'activité de l'objet.

Le fait que la programmation soit ouverte (contrôle du service des requêtes) se révèle particulièrement important pour la construction d'un service de migration puissant et flexible. Cet aspect de programmation ouvert, permet aisément d'étendre le modèle de ProActive pour des besoins spécifiques [13].

2.12 Notion de Groupe :

2.12.1 Principes

Le système de communication de groupe repose sur le mécanisme élémentaire d'invocation distante et asynchrone de méthodes. Comme l'ensemble de la librairie, ce mécanisme est mis en application en utilisant une version standard de Java.
Le mécanisme de groupe est indépendant de la plateforme. Il doit être considéré comme une réplique de plusieurs invocations à distance de méthode vers des objets actifs. Naturellement, le but est d'incorporer quelques optimisations à l'exécution, de façon à réaliser de meilleures exécutions qu'un accomplissement séquentiel de *n* appels de méthode à distance. De cette façon, notre mécanisme est la généralisation du mécanisme d'appel de méthode asynchrone sur des objets distants.
La disponibilité d'un tel mécanisme de communication de groupes simplifie la programmation des applications en regroupant les activités semblables fonctionnant en parallèle. En effet, du point de vue de la programmation, utiliser un groupe d'objets du même type, appelé *groupe typé*, prend exactement la même forme que l'utilisation d'un simple objet de ce type. Ceci est possible grâce à des techniques de réification : la classe d'un objet que nous voulons rendre actif et accessible à distance est étendue au moment de l'exécution, et les appels de méthode sont réifiés.

D'une manière transparente, les appels de méthode dirigés vers un objet actif sont exécutés au travers d'un stub 2 qui est d'un type compatible avec l'objet original. Le rôle du stub est de contrôler l'appel en lui appliquant la sémantique exigée : s'il s'agit d'un appel vers un objet actif distant simple, alors l'invocation à distance asynchrone standard est appliquée ; si l'appel est dirigé vers un groupe d'objets, alors la sémantique des communications de groupes est appliquée comme nous le verrons dans le reste de cette section.

2.12.2 Création d'un groupe

Les groupes sont créés en utilisant la méthode statique :

```
ProActiveGroup.newGroup("ClassName", paramètres[], noeuds[]);
```

La superclasse commune à tous les membres du groupe doit être indiquée à la création du groupe, et lui donne ainsi un type minimal. Les groupes peuvent être créés vides, puis remplis par des objets actifs déjà existants. Des groupes non-vides peuvent aussi être construits en utilisant deux paramètres supplémentaires : une liste de paramètres requis pour la construction des membres du groupe et la liste des noeuds où ils seront créés. Le nième objet actif est créé avec les nièmes paramètres sur le nième nœud.
Dans ce cas, le groupe est créé et les objets actifs sont construits puis immédiatement inclus dans le groupe. Voici un exemple :

```
class A {
public A()
public void foo () {...}
public V bar (B b) {...}
}
// Pré-construction de paramètres pour la creation d'un groupe
Object[][] params = { {...} , {...} , ... };
// Noeuds sur lesquels seront créés les objets actifs
Node[] nodes = { ... , ... , ... };
// Création 1:
// Création d'un groupe vide de type "A"
A ag = (A) ProActiveGroup.newGroup("A");

// Création 2:
// Un groupe de type "A" et ses membres sont créés en même temps
A ag2 = (A) ProActiveGroup.newGroup("A", params, nodes);
```

Des éléments ne peuvent être inclus dans un groupe que si leur type est compatible avec la classe spécifiée à la création du groupe. Par exemple, un objet de classe B (B étendant A) peut être inclus dans le groupe. Cependant, étant basées sur le type de A, seules les méthodes définies dans la classe A peuvent être appelées sur le groupe, mais notons que la redéfinition de méthode va fonctionner normalement.
La limitation principale de la construction de groupe est que la classe indiquée au groupe doit être ***réifiable***, selon les contraintes imposées par le protocole à méta objet de *ProActive* : le type ne doit pas être un type primitif (int, double, boolean,...), ni une classe final. Dans ces cas, on ne peut pas créer de groupe d'objet [4].

2.12.3 Représentations et manipulation de groupes

La représentation typée des groupes correspond à la vue fonctionnelle des groupes d'objets. Afin de fournir une gestion dynamique des groupes, une deuxième (et complémentaire) représentation d'un groupe a été conçue. Cette seconde représentation suit un modèle plus standard : l'interface Group étend l'interface Collection de Java qui fournit des méthodes telles que : ***add, remove, size***, ... Cette gestion de groupes comporte une sémantique simple et classique (ajouter dans le groupe, enlever le nième élément, ...) qui fournit une propriété de rang des éléments au sein d'un groupe. Les méthodes de gestion d'un groupe ne sont pas disponibles dans la *représentation typée* mais dans la *représentation de groupe*. La double représentation est un choix de conception. Il existe deux représentations complémentaires, l'une pour l'usage fonctionnel et l'autre pour la gestion dynamique. Au niveau de l'implémentation, les développeurs de ProActive ont pris soin de maintenir une cohérence forte entre les deux représentations d'un même groupe. Les modifications faites sous une forme sont instantanément reportées sous l'autre forme. Pour alterner d'une forme à l'autre deux méthodes sont définies [8,4] :

La méthode statique ***getGroup*** de la classe ***ProActiveGroup*** retourne la représentation de groupe à partir d'une représentation typée. La méthode ***getGroupByType*** définie dans l'interface Group fournit l'opération inverse.

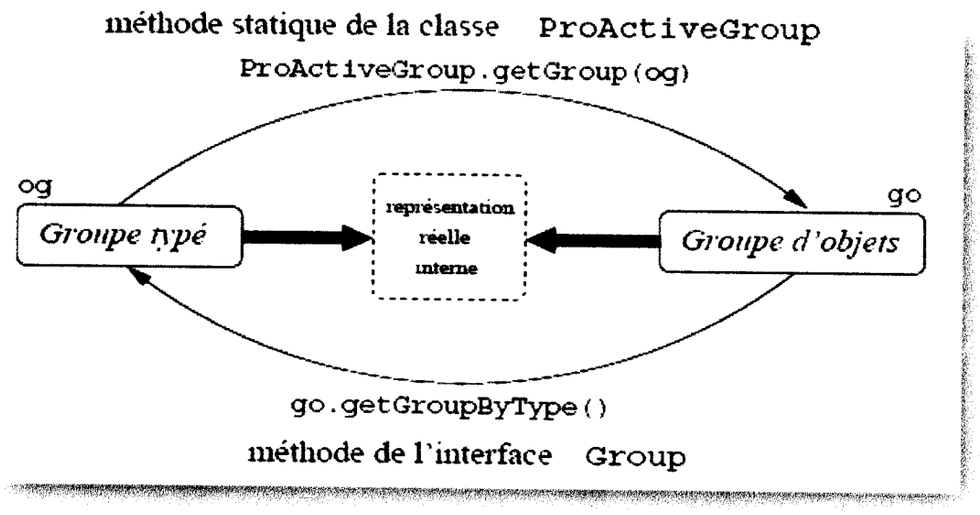

Fig. 2.5 Double représentation des groupes[8].

Voici un exemple de l'emploi de chaque représentation d'un groupe :

```
// Creation d'un objet Java standard et de deux objets actifs
A a1 = new A();
A a2 = (A) ProActive.newActive("A", paramsA[], node);
B b = (B) ProActive.newActive("B", paramsB[], node);

// Notons que B etend A
// Pour la gestion d'un groupe, on obtient la représentation
// de groupe à partir d'un groupe typé
Group gA = ProActiveGroup.getGroup(ag);
// On ajoute les objets au groupe
gA.add(a1);
gA.add(a2);
gA.add(b);
// Une nouvelle référence vers le groupe typé peut être obtenue
A ag_new = (A) gA.getGroupByType();
```

Notons que les groupes ne contiennent pas nécessairement que des objets actifs, mais peuvent contenir des objets standard de Java et des groupes typés (d'où l'obtention de groupes hiérarchiques). La seule restriction est qu'ils soient de classe compatible avec la classe du groupe. La section suivante examinera les implications de tels groupes hétérogènes dans la gestion des communications vers les éléments du groupe.

2.12.4. Communication de groupe

L'invocation d'une méthode sur un groupe a une syntaxe identique à une invocation de méthode sur un objet Java :

```
// Une communication de groupe
 ag.foo();
```

Bien sûr, un appel de ce type a une sémantique différente : l'appel de méthode est rendu asynchrone et est propagé vers tous les membres du groupe. Un appel de méthode sur un groupe est un appel de méthode sur chaque membre du groupe. Ainsi, si un membre est un objet actif, la sémantique de communication de *ProActive* sera utilisée, s'il s'agit d'un objet Java, la sémantique sera celle d'un appel de méthode classique.
Par défaut, les paramètres de la méthode invoquée sont diffusés à tous les membres du groupe ***(broadcast)***.
Il est également possible, grâce à des méthodes statiques, de changer le comportement des groupes pour que les paramètres soient distribués selon les membres ***(scatter)*** et non plus diffusés : pour distribuer les données à travers une communication de groupe, il suffit de rassembler ces données au sein d'un groupe et de le passer en paramètre à un appel de méthode [8].

Voici un exemple :

```
// Création du groupe de parametres
B bg = (B) ProActiveGroup.newGroup("B", {{...},{...},...} ,
{...,...,...});
// bg est envoyé à tous les membres de ag
ag.bar(bg);
// Changement du mode de communication de ag (distribution)
ProActiveGroup.setScatterGroup(bg);
// Chaque membre de bg est envoyé à un membre de ag
ag.bar(bg);
```

3.12.5 Résultat d'une communication de groupe

La particularité de notre mécanisme de communication est que le résultat de la communication d'un groupe typé est un groupe typé. Ce groupe résultat est construit dynamiquement et de façon transparente au moment de l'invocation de la méthode, avec un futur pour chaque réponse attendue. Le groupe résultat est mis à jour au fur et à mesure que les réponses arrivent dans le contexte de l'appelant. Toutefois,il peut être instantanément utilisé pour lancer un appel de méthode sachant que le mécanisme d'*attente par nécessité* entre en jeu : si tous les résultats ne sont pas encore arrivés, l'appel de méthode se fera automatiquement au moment de leurs retours.
Dans le code ci-dessous, un nouvel appel de méthode de f1() est automatiquement déclenché dès qu'une réponse de l'appel **vg = ag.bar(...)** reviendra dans le groupe **vg** :

```
// Un appel de méthode qui renvoie un résultat
V vg = ag.bar(...);
// vg est un groupe typé de type "V"
// L'opération suivante est aussi une communication de groupe sur
// les résultats de l'appel précédent.
vg.f1();
```

Comme la montre la figure si dessous, le placement des éléments dans le groupe est une propriété conservée à travers un appel de méthode : le résultat de l'appel de méthode appliquée au nième membre d'un groupe est stocké à la nième place dans le groupe résultat.

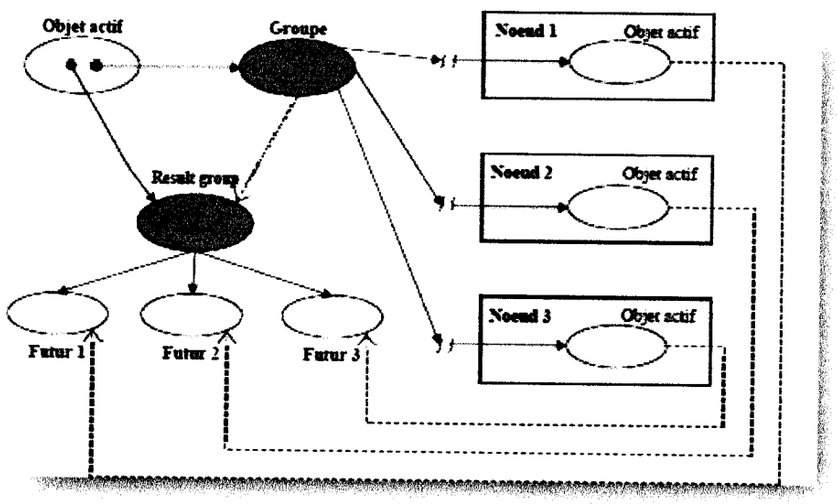

Fig. 2.3 Communication de groupe avec groupe résultat[8].

Des groupes dont le type est basé sur des classes finales ou des types primitifs ne peuvent pas être construits. Par conséquent, la construction dynamique d'un groupe de résultats est également limitée :
Seules les méthodes dont le type de retour est, soit vide, soit de type ***réifiable*** peuvent être invoquées sur un groupe d'objets ; autrement, elles lèveront une exception au moment de l'exécution parce que la construction transparente d'un groupe de futurs de types non ***réifiable*** a échoué.

3.12.6 Synchronisations sur des groupes résultats

Pour profiter du modèle d'appel distant et asynchrone de *ProActive*, quelques nouveaux mécanismes de synchronisation ont été ajoutés. Des méthodes définies sur l'interface Group permettent d'exécuter diverses formes de ***synchronisation (waitOne, waitN, waitAll, waitTheNth, waitAndGet, ...)***[4].

Par exemple :

```
// Une méthode appelée sur un groupe type
V vg = ag.bar(...);
// On accède à la représentation Group de vg
Group gV = ProActiveGroup.getGroup(vg);
// Pour attendre et retourner le premier membre de vg
V v = (V) gV.waitAndGetOne();
// Pour attendre que tous les membres de vg soient arrivés
gV.waitAll();
```

2.12.7 Mécanismes d'optimisation

Une implémentation naïve apportait déjà des gains de performances. Un appel de méthode sur un groupe de *n* objets est plus rapide que le contact des *n* objets de façon individuelle. Cette première amélioration provient de l'économie de plusieurs réifications d'appels de méthode. Cette opération du méta-niveau construit un objet représentant l'appel de méthode. Lors d'un appel de groupe un seul objet de ce type est construit [4].

2.12.8 Multithreading

L'utilisation de plusieurs fils d'exécution (*threads*) permet l'envoi simultané des messages vers chaque destinataire. Les temps des rendez-vous RMI sont ainsi recouverts et non pas cumulés comme cela aurait été le cas si les appels avaient été successifs. Pour conserver la sémantique de *ProActive* une barrière de synchronisation assure que toutes les requêtes ont été transmises aux objets distants et placées dans leur file d'attente avant de passer à l'instruction suivant une communication de groupe [4].

2.12.9 Sérialisation unique

Le protocole RMI se charge de transmettre les paramètres de l'appel à tous les membres en les sérialisant puis en les transmettant sur le réseau. La sérialisation est un processus particulièrement lent de Java

Dans le cas d'une diffusion des mêmes paramètres à tous les objets (*broadcast*), ces paramètres seront sérialisés par chaque thread. Pour éviter ce gaspillage de ressources, une sérialisation unique des paramètres de l'appel est faite par le mécanisme de communication de groupes avant que les appels ne soient délégués à RMI [4].

2.13 Mécanisme de déploiement

ProActive fournit une fonctionnalité très pratique pour répandre rapidement des objets actifs sur de nombreuses machines. On utilise un fichier *XML* pour décrire le déploiement de l'application. On définit à l'intérieur de ce fichier tous les paramètres nécessaires à ProActive pour créer les machines virtuelles et les noeuds sur les ordinateurs distants.
Le fichier commence par un *XML Namespaces* (*xmlns*), un espace de nommage, associé à une URL pour l'identifier, et par une adresse de *Schéma* définissant la grammaire *XML* utilisé [9] :

```
<ProActiveDescriptor xmlns:xsi="http://www.w3.org/2001/
XMLSchemainstance"
xsi:noNamespaceSchemaLocation="DescriptorSchema.xsd">
```

Avant d'expliquer ce fichier en détail, on peut représenter une hiérarchie des balises de déploiement :

```
<ProActiveDescriptor ...>
<componentDefinition>
<virtualNodesDefinition> <!--details le nom des noeuds virtuels-->
</virtualNodesDefinition> <!--et ses propriétés-->
</componentDefinition>
<deployment>
<mapping>
<map>
<jvmSet> <!--associer des noms de JVM à chaque-->
</jvmSet> <!--noeud virtuel-->
</map>
</mapping>
<jvms> <!--chaque JVM est associé à un nom de-->
</jvms> <!--processus-->
</deployment>
<infrastructure>
<processes>
<processDefinition> <!--indication de l'adresse des classes-->
</jvmProcess> <!--des programmes et des fichiers-->
<classpath> <!--pour le fonctionnement des JVM-->
</classpath>
<javaPath>
</javaPath>
<policyFile>
</policyFile>
<log4jpropertiesFile>
</log4jpropertiesFile>
</jvmProcess>
</processDefinition>
<processDefinition> <!--chaque JVM est associé à une adresse-->
<rshProcess> <!--de machine et à un protocol d'accès-->
</rshProcess> <!--à distance (ici rsh)-->
</processDefinition>
</processes>
</infrastructure>
</ProActiveDescriptor>
```

Avec la balise **<componentDefinition>** on commence par définir un ou plusieurs *nœuds virtuels* auquel on attachera un groupe de noeuds à créer.

```
<componentDefinition>
<virtualNodesDefinition>
<virtualNode name="NoeudSalut" property="multiple"/>
</virtualNodesDefinition>
</componentDefinition>
```

La propriété **multiple** permet d'indiquer que plusieurs noeuds seront associés à ce noeud virtuel.

Puis avec la balise **<deployment>** on définit les machines virtuelles Java associé à ce nœud virtuel (**"Jvm1"...**), sur lesquelles un noeud sera créé. On associe à chaque machine virtuelle un processus s'occupant de la connexion à une machine et de la création d'un nœud (**"rshProcess1"...**).

```
<deployment>
<mapping> <!--définition des JVM associées aux noeuds virtuels-->
<map virtualNode="NoeudSalut">
<jvmSet>
<vmName value="Jvm1"/>
<vmName value="Jvm2"/>
.....
<vmName value="Jvm7"/>
<vmName value="Jvm8"/>
</jvmSet>
</map>
</mapping>
<jvms> <!--définition des processus s'occupant de la connexion-->
<jvm name="Jvm1"> <!--à une machine et de la création d'un noeud-->
<creation>
<processReference refid="rshProcess1"/>
</creation>
</jvm>
<jvm name="Jvm2">
<creation>
<processReference refid="rshProcess2"/>
</creation>
</jvm>
.....
<jvm name="Jvm7">
<creation>
<processReference refid="rshProcess7"/>
</creation>
</jvm>
<jvm name="Jvm8">
<creation>
<processReference refid="rshProcess8"/>
</creation>
</jvm>
</jvms>
</deployment>
```

La balise **<infrastructure>** permet d'abord d'associer à un processus d'exécution d'une JVM **"MyJvmProcess"** l'adresse de l'interpréteur Java et de tous les fichiers nécessaires au fonctionnement d'une machine virtuelle et de son nœud : classes, jar, politique de sécurité et journalisation des erreurs. Ensuite, à chaque processus de création des machines virtuelles (par exemple **"rshProcess1"**) est associé le programme de connexion (utilisant un protocole de connexion à distance : ***rsh, ssh*** ...) avec le nom de la machine à atteindre, et le processus d'exécution d'une JVM permettant de lancer la machine virtuelle et de créer le nœud.

```
<!--dernière partie du fichier xml pour définir les processus de-->
<!--création des JVM-->
<infrastructure>
<processes> <!--définition du processus d'exécution d'une JVM-->
<processDefinition id="MyJvmProcess">
<jvmProcess
class="org.objectweb.proactive.core.process.JVMNodeProcess">
<classpath>
<absolutePath
value="/usr/users/staginfo/bezzine/ProActive/classes"/>
<absolutePath
value="/usr/users/staginfo/bezzine/ProActive/ProActive.jar"/>
<absolutePath
value="/usr/users/staginfo/bezzine/ProActive/lib/bcel.jar"/>
.....
<absolutePath
value="/usr/users/staginfo/bezzine/ProActive/lib/jsch.jar"/>
</classpath>
<javaPath>
<absolutePath value="/usr/lib/jre/bin/java"/>
</javaPath>
<policyFile>
<absolutePath
value="/usr/users/staginfo/bezzine/ProActive/scripts
/proactive.java.policy"/>
</policyFile>
<log4jpropertiesFile>
<absolutePath
value="/usr/users/staginfo/bezzine/ProActive/scripts/proactive-log4j"/>
</log4jpropertiesFile>
</jvmProcess>
</processDefinition>
<processDefinition id="rshProcess1"> <!--définition du processus-->
<rshProcess <!--de création d'une JVM distante-->
class="org.objectweb.proactive.core.process.rsh.RSHJVMProcess"
hostname="taliani">
<processReference refid=" MyJvmProcess"/>
</rshProcess>
</processDefinition>
```

```
.....
<processDefinition id="rshProcess8">
<rshProcess
class="org.objectweb.proactive.core.process.rsh.RSHJVMProcess"
hostname="sh08">
<processReference refid=" MyJvmProcess"/>
</rshProcess>
</processDefinition>
</processes>
</infrastructure>
```

Il faut faire attention à un point important en ce qui concerne ce fichier *XML*. En principe une machine virtuelle recherche des classes en se référant au ***classpath*** du système d'exploitation. Mais lorsque des objets actifs créés sur un nœud ont besoin des classes d'un projet pour instancier des objets, l'adresse du projet doit être précisée dans le ***classpath*** du processus de création de la machine virtuelle et du nœud du descripteur de déploiement. En fait la machine virtuelle créée à distance par ProActive ne se réfère qu'aux fichiers indiqués dans le descripteur *XML* sans tenir compte du ***classpath*** de la machine hôte.

12.14 Exemple de déploiement d'objets ProActive

Nous allons voir dans un exemple comment utiliser le descripteur de déploiement pour créer des objets actifs distants et des groupes.
On crée tout d'abord la classe **Salut** permettant de créer les futurs objets actifs, cette classe possède une méthode retournant la chaîne "**Salut de :** " concaténée avec l'adresse de la machine envoyant le message :

```
//la classe Salut permettant la creation d'objets actifs
public class Salut {
//constructeur vide pour creer des objets actifs
public Salut() {}
//une methode retournant un objet réifiable
public ContainerString saysalut() {
return new ContainerString ("Salut de : "
+ ProActive.getBodyOnThis().getNodeURL());
}
}
```

On utilise **ContainerString** comme objet de retour et non **String** car ce dernier n'est pas *réifiable* (la classe **String** possède l'attribut *final*), donc on ne pourrait pas en faire un objet *futur*et utiliser la communication de groupe. La classe **ContainerString** peut être écrite très simplement comme suit :

```
//classe permettant la création d'objets réifiables de type String
public class ContainerString implements Serializable{
private String string;
public ContainerString() {}
public ContainerString (String string){
this.string=string;
}
public String getString(){
return(string);
}
}
```

Puis dans une autre partie de l'application on va déployer et utiliser des objets actifs de la classe **Salut** sur un cluster. On associe un fichier *XML* à une variable ProActive **(pad = getProactiveDescriptor())**.
Ensuite on active la création des nœuds distants définis dans le descripteur de déploiement **(pad.activateMappings())**.
On récupère le *noeud virtuel* **(pad.getVirtualNode("NoeudSalut"))** qui vient d'être créé. On l'utilise comme argument pour créer un groupe d'objets actifs **(ProActiveGroup.newGroup(...))** : un objet actif est créé sur chaque nœud associé au *nœud virtuel*. Dans le descripteur de déploiement huit nœuds sont associés au *nœud virtuel* obtenu, donc huit objets actifs sont créés. Dans cet exemple on crée aussi un objet passif et un objet actif en local, puis on les ajoute au groupe **(groupStdSalut.add(...))** grâce à la représentation standard du groupe **(groupStdSalut)**.

Après avoir lancé la méthode **saysalut()** sur le groupe, on obtient un groupe de messages résultats, on récupère chaque résultat du groupe de résultat avec la méthode **get()**, ensuite avec la méthode **getString()** de **ContainerString** on peut obtenir le vrai message de chaque message résultat (donc de chaque noeud dans cet exemple).

Enfin la méthode **killall(false)** associé au descripteur ProActive **pad** permet de détruire tous les nœuds et machine virtuelles créés.

```
//extrait de code d'une application deployant et utilisant des
//objets actifs Salut
try {
//déploiement des noeuds décrits dans le fichier Salut.xml
ProActiveDescriptor pad = ProActive.getProactiveDescriptor(
"/usr/users/staginfo/bezzine/ProActive/descriptors/Salut.xml");
pad.activateMappings();
VirtualNode virtuelNoeudSalut = pad.getVirtualNode("NoeudSalut");
//creation d'un groupe d'objets actifs Salut sur les noeuds
//associés au noeud virtuel VirtuelNoeudSalut
Salut groupSalut = (Salut) ProActiveGroup.newGroup(
Salut.class.getName(), new Object[] {}, virtuelNoeudSalut);
//création d'un objet classique sur la machine local
Salut salut1 = new Salut();
//création d'un objet actif sur la machine local
Salut salut2 = (Salut) ProActive.newActive(
Salut.class.getName(), new Object[] {});
//création de la représentation standard du groupe pour ajouter 2
//objets au groupe
Group groupStdSalut = ProActiveGroup.getGroup(groupSalut);
groupStdSalut.add(salut1);
groupStdSalut.add(salut2);
//méthode saysalut() lancée sur le groupe et récupération des
//resultats
ContainerString groupMessageRes = GroupSalut.saysalut();
Group groupStdMessageRes =
ProActiveGroup.getGroup(groupMessageRes);
for(int i=0;i< groupStdMessageRes.size();i++) {
System.out.println(
((ContainerString) groupStdMessageRes.get(i)).getString());
}
//destruction des noeuds et des objets associés
pad.killall(false);
System.exit(1);
} catch (Exception e) {
System.err.println("Error: " + e.getMessage());
e.printStackTrace();
}
```

Voici le résultat affiché sur la console du poste utilisateur :

```
Salut de : //snake/NoeudSalut-1276061078
Salut de : //taliani/NoeudSalut-1197957052
Salut de : //sh06/NoeudSalut-1206421527
Salut de : //sh07/NoeudSalut-1199880796
Salut de : //sh04/NoeudSalut-1204497783
Salut de : //sh05/NoeudSalut-1220657237
Salut de : //sh08/NoeudSalut-1550387044
Salut de : //sh02/NoeudSalut-1562699009
Salut de : LOCAL
Salut de : //sh00/Node-371497875
```

12.15 Bilan :

La documentation ProActive permet assez facilement de commencer à installer et à utiliser ProActive sur un exemple simple fourni, mais elle n'est malheureusement pas assez détaillée pour maîtriser rapidement des opérations plus complexes comme les communications de groupes et le fichier de déploiement. Il faut passer beaucoup de temps à expérimenter soi-même sur de petits programmes pour vraiment bien comprendre le fonctionnement de ProActive.

Une fois assimilé, ProActive est très pratique pour distribuer des calculs et déployer des objets sur un grand nombre de machines, mais malheureusement il utilise des protocoles comme ssh, rsh qui ne sont pas implémentés dans l'environnement WINDOWS c'est se que nous a obliger de choisir comme plateforme le system UNIX .

Chapitre 3 :

Conception et Modélisation

3.1 Introduction :

Dans cette partie, nous allons présenter les différentes étapes de développement de notre application. Il est possible de mettre à profit des outils rigoureux pour aborder l'analyse du projet. Ces outils font partie de ceux que l'on utilise en génie logiciel. A ce titre on étudie donc les cas d'utilisation (*use case*), les *problèmes frames* pour enfin aboutir à une modélisation UML du problème abordé, En effet c'est un langage conçu précisément des applications orienté objet dont la nôtre.

3.2 Cahier de Charge :

3.2.1 Présentation de l'environnement :

Notre application que nous avons nommée ***UnivGrid*** (grid university) se compose essentiellement en deux parties, la première qui est l'architecture Client-Serveur qui utilise la technologie JSP (un site web) entre le client et le serveur, et la seconde partie est l'architecture Serveur-Grille qui utilise l'API ProActive pour la communication entre le Serveur et les différentes machines inscrites sur la grille effectuant des calculs.

3.2.2 Pourquoi une telle application ?

On trouve dans la majorité des entreprises et dans les universités des ordinateurs non utilisés, alors qu'elles font souvent des calculs énormes et qui prend un temps de traitement très long sur une seule machine. Nous avons pensé à exploiter ces ressources non utilisées pour distribuer un grand nombre de calculs indépendants afin de donner des résultats sur un temps très petit par rapport à un traitement sur une seule machine.

Cette exploitation va se faire à travers une grille de calcul dont le mécanisme de communication se fait au moyen d'un middleware : Parmi les middleware existants (Globus, Condor, JavaSpaces ,etc.) nous avons choisi ProActive pour les raisons suivantes :

a) Division du problème :

ProActive permet de diviser les tâches en plusieurs sous taches ; ceci est réalisé par la création d'un ensemble d'objets actifs sur des machines différentes, à chaque objet actif on attribue des sous taches .

b) Conserver la flexibilité initiale :

- **Au niveau de l'utilisation des objets :**il faut garder une certaine cohérence au niveau du code et on souhaite de préférence utiliser des objets distants, effectuer des appels de méthodes comme si les objets se trouvent sur la machine locale ,en java RMI fournit cette possibilité.
- **Au niveau de la compilation :**il ne faut pas recompiler tout le code à chaque modification des paramètres ,ou lorsqu'on veut changer le nombre de sous-domaines, le nombre d'unités de calcul, le mode de connexion entre les machines. Toutes ces données doivent être stockées dans un fichier de description externe.
- **Au niveau de déploiement :** l'utilisateur ne doit rien avoir à entrer, sauf les paramètres initial du calcul bien sûr, en plus de ça si votre système contient n machines il est hors de question de lancer le script sur toute les machines ainsi quelques unes sont gardées afin de remplacer un pc qui a peut tomber en panne.
- **Réduire les temps de communication :** En plus de condenser les données qui transitent, il faut la possibilité de lancer des appels de méthodes asynchrones et récupérer les résultats plus tard, et lorsqu'une même opération doit être lancée sur plusieurs objets en même temps il faut mettre en œuvre une communication de groupe.

Pour toutes ces raisons, nous avons choisi d'utiliser ProActive afin de permette de disperser l'exécution d'un programmes sur plusieurs machines en effectuant la communication selon différents protocoles (SSH,RLOGIN,RSH…).

3.2.3 Objectif de l'application

L'Objectif de notre projet c'est d'étudier l'architecture de grille en utilisant la bibliothèque ProActive . Cela permettra de réaliser une application qui utilise la grille de calcul en toute transparence pour le client, et qui va donner des résultats avec des très grandes précisions en un temps très court.

Dans notre application, on a développé un calcul de grille sur deux exemples :

a) Formule de Pi (BBP):

Cette formule à l'incroyable propriété d'autoriser le calcul des décimales de Pi indépendamment les unes des autres.

La formule BBP se présente comme suite :

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right)$$

b) Formule de Bessel :

Cette fonction est une fonction mathématique obtenue à partir des fonctions de Bessel. Elle est utilisée en électromagnétisme pour étudier :

* Les solutions des équations de Maxwell dans des domaines conducteurs de forme cylindrique.

* Les ondes électromagnétiques dans un guide cylindrique (Guide d'onde).

* Modélisation du rayonnement des antennes.

* Au premier ordre (n=1) cette fonction est utilisée pour étudier une antenne conique ou parabolique.

* L'étude d'instruments optiques.

La formule se présente comme suite :

$$f(x) = \propto_0 \sum_{k=0}^{\infty} (-1)^k . \frac{x^{2k}}{2k!} + \propto_1 \sum_{k=0}^{\infty} (-1)^{k+1} . \frac{x^{2k+1}}{(2k+1)!}$$

3.3 Génie Logiciel appliqué à l'analyse

Basé sur les études et les analyses faites précédemment, nous allons utiliser le modèle UML pour la modélisation et conception de notre application puisqu'il est un langage graphique de modélisation des données et des traitements.

Une des caractéristiques importantes d'UML est qu'il permet de représenter, les différents diagrammes.

Un diagramme UML est une représentation graphique, qui s'intéresse à un aspect précis du modèle, chaque type de diagramme UML possède une structure (les types des éléments de modélisation qui le composent sont prédéfinis) et véhicule une sémantique précise (il offre toujours la même vue d'un système).

Combinés, les différents types de diagrammes UML offrent une vue complète des aspects statiques et dynamiques d'un système. Les diagrammes permettent donc d'inspecter un modèle selon différentes perspectives et guident l'utilisation des éléments de modélisation (les concepts objet), car ils possèdent une structure.

Pour ne pas alourdir le rapport et d'entrer dans les détails techniques, nous ne présenterons que quelques diagrammes que nous jugeons utiles pour comprendre le projet.

3.3.1 Diagramme de cas d'utilisation :

Les diagrammes de cas d'utilisation sont des diagrammes UML utilisés pour donner une vision globale du comportement fonctionnel d'un système logiciel. Un cas d'utilisation représente une unité discrète d'interaction entre un utilisateur (humain ou machine) et un système. Il est une unité significative de travail. Dans un diagramme de cas d'utilisation, les utilisateurs sont appelés acteurs, ils interagissent avec les cas d'utilisation.

a) Les Acteurs :

Ce sont des personnes ou systèmes extérieurs à l'application, et qui interagissent avec elle.

On définie pour notre application les acteurs suivant :

- Utilisateur : Acteur secondaire, c'est la personne qui consulte le site web, et qui peut a tout moment s'enregistrer ou s'identifier pour devenir un Client
- Client : Acteur principal, la personne désirant effectuée les calculs sur la grille.
- Serveur : Acteur principal, c'est la machine qui distribue et lance les calculs sur la grille après authentification du client.
- Nœud : Acteur principal, la machine qui effectue le calcul.
  (Plusieurs Nœud forment la grille de calcul).

b) Les Actions:

Elles représentent une fonctionnalité (un objectif à atteindre) du système à construire. Ils sont en relation avec des acteurs et d'autres cas d'utilisation.

A partir des Acteurs cité dessus les cas d'utilisations sont identifiés comme suite :

- Consultation : un utilisateur peut consulter le site (documentations)
- Inscription : Inscription de l'utilisateur en fournissant des informations et les stocker dans la base de donnée (informations personnelles + mot de passe).
- Identification : l'utilisateur introduit son login, mot de passe et les envoie au système pour devenir un client.
- Authentification : le système vérifie le login et le mot de passe fourni par le client (stocker dans la base de données).
- Fournir paramètres de calcule : le client introduit les données nécessaires pour le calcul.
- Distribution et lancement de calcul : le serveur distribue les paramètres de calcul à la grille et ordonne le lancement de ce dernier.
- Effectuer les calculs et génération de résultats : l'ensemble de postes formant la grille effectuent les calculs et génère chacun un bout de résultat.
- Récupération de résultats : le serveur récupère tous les bouts de calculs et génère un résultat final.
- Fournir le résultat : le serveur fourni le résultat final au client.

3. 3. 2 Description des scénarios :

Suite au cas d'utilisation nous allons présenter quelques scénarios que nous jugeons intéressant.

a) **Identification :** Un utilisateur s'identifie auprès du système.

-Pré condition: L'utilisateur est sur le site web (en tapant l'URL du site sur un navigateur).

-Intention dans le contexte : L'intention de l'utilisateur est de s'identifier auprès d'UnivGrid

-Succès du scénario :

1. L'utilisateur tape son login et son mot de passe et les envoie au système.
2. Le système vérifie l'existence du login.
3. Le système vérifie la validité du mot de passe.
4. Le système informe l'utilisateur qu'il est correctement identifié.

-Extensions :

1. Le système est hors d'atteinte:
Le cas d'utilisation se termine avec erreur.

2. Le login fourni n'est pas reconnu par le système:
Le système informe l'utilisateur.
Le système demande à l'utilisateur de fournir un login valide.
Retour à l'étape 1.

3. Le mot de passe fourni n'est pas correct:
Le système informe l'utilisateur.
Le système demande à l'utilisateur de fournir un mot de passe valide.
Retour à l'étape 1.

***b)* Lancement d'un calcul :** Un utilisateur lance un calcul sur la grille.

-Pré condition: L'utilisateur est identifié auprès du système.

-Intention dans le contexte : L'intention de l'utilisateur est de lancer un calcul sur la grille de calcul.

-Succès du scénario :

1. L'utilisateur fournit au système les données initiales au lancement du calcul à travers un formulaire que le système lui affiche.
2. Le système "découpe" le calcul.
3. Le système sélectionne les nœuds pour effectuer les calculs.
4. Le système lance les calculs sur les nœuds sélectionnés.

-Extensions :

1. Les données spécifiées ne sont pas valides:
   Le système informe l'utilisateur.
   Le système demande à l'utilisateur d'entrer à nouveau les données.
   Retour à l'étape 1.
2. Le système ne trouve pas de nœuds aptes à effectuer le calcul demandé:
   Le système informe l'utilisateur.
   Le cas d'utilisation se termine avec erreur.

c) ***Récupération d'un résultat :*** Un utilisateur récupère le résultat d'un calcul qu'il a précédemment lancé sur la grille.

-Pré condition: L'utilisateur est sur le site web (en tapent l'URL du site sur un navigateur).

-Intention dans le contexte : L'intention de l'utilisateur est de s'identifier auprès d'UnivGrid

-Succès du scénario :

1. Le système récupère les résultats des différents noeuds ayant participé au calcul.
2. Le système regroupe tous les sous résultats et crée le résultat final.
3. Le système fait parvenir le résultat final à l'utilisateur.

-Extensions :

1. Tous les sous résultats ne sont pas accessibles:
   Le système informe l'utilisateur.
   Le cas d'utilisation se termine avec erreur.

3. 3. 3 Diagramme de séquences :

Avec les diagrammes de séquences, l'UML fournit un moyen graphique pour représenter les interactions entre objets à travers le temps. Ces diagrammes montrent typiquement un utilisateur ou un acteur et les objets et composants avec lesquels ils

interagissent au cours de l'exécution du cas d'utilisation. Un diagramme de séquence représente en général un seul 'scénario' de Cas d'Utilisation ou flux d'événements.

Les diagrammes de séquence sont une excellente façon pour documenter les scénarios d'utilisation, identifier les objets requis tôt dans l'analyse et vérifier leur utilisation plus tard dans la conception.

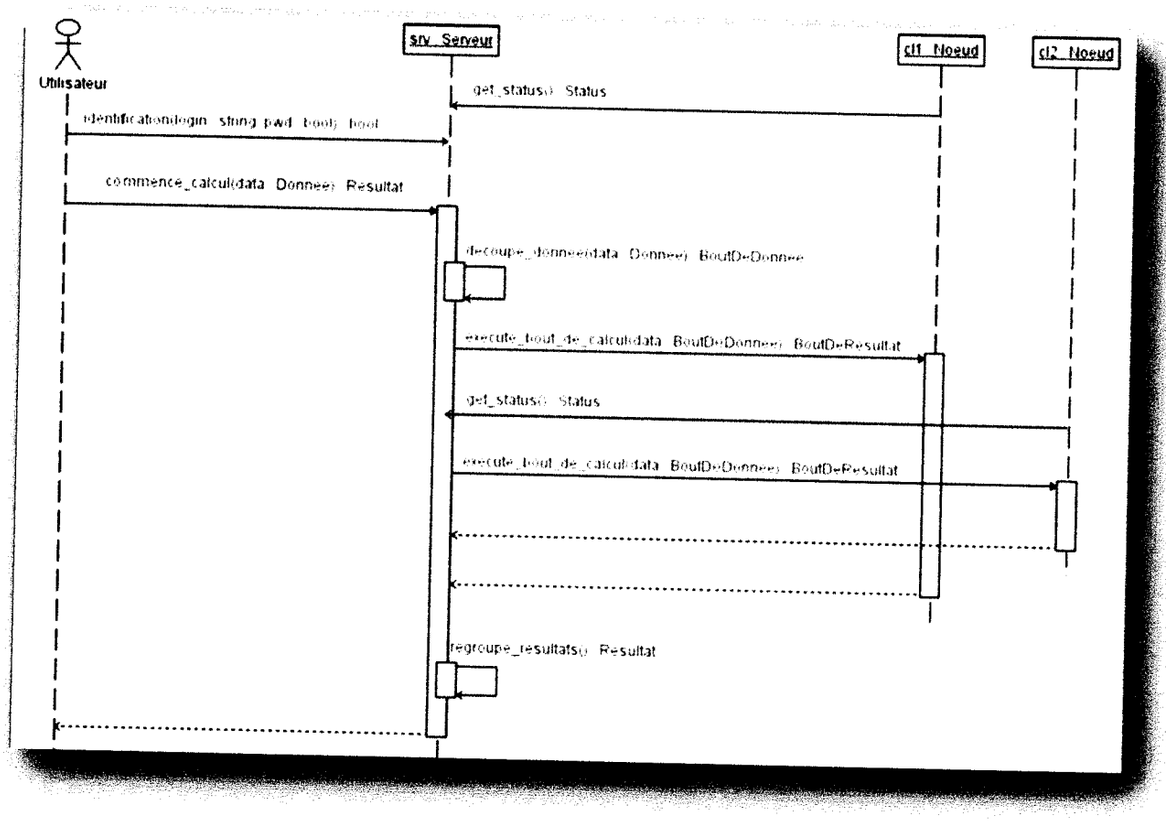

Fig. 3.1 : *Diagramme de séquence*

3. 3. 4 Diagramme de classes :

Le diagramme de classes est un schéma utilisé en génie logiciel pour présenter les classes et les interfaces d'un système ainsi que les différentes relations entre celles-ci. Ce diagramme fait partie de la partie statique d'UML car il fait abstraction des aspects temporels et dynamiques.
Une classe décrit les responsabilités, le comportement et le type d'un ensemble d'objets. Les éléments de cet ensemble sont les instances de la classe.
Le diagramme de classes a pour objet de mettre en évidence les classes d'un système avec les relations qui les associent.

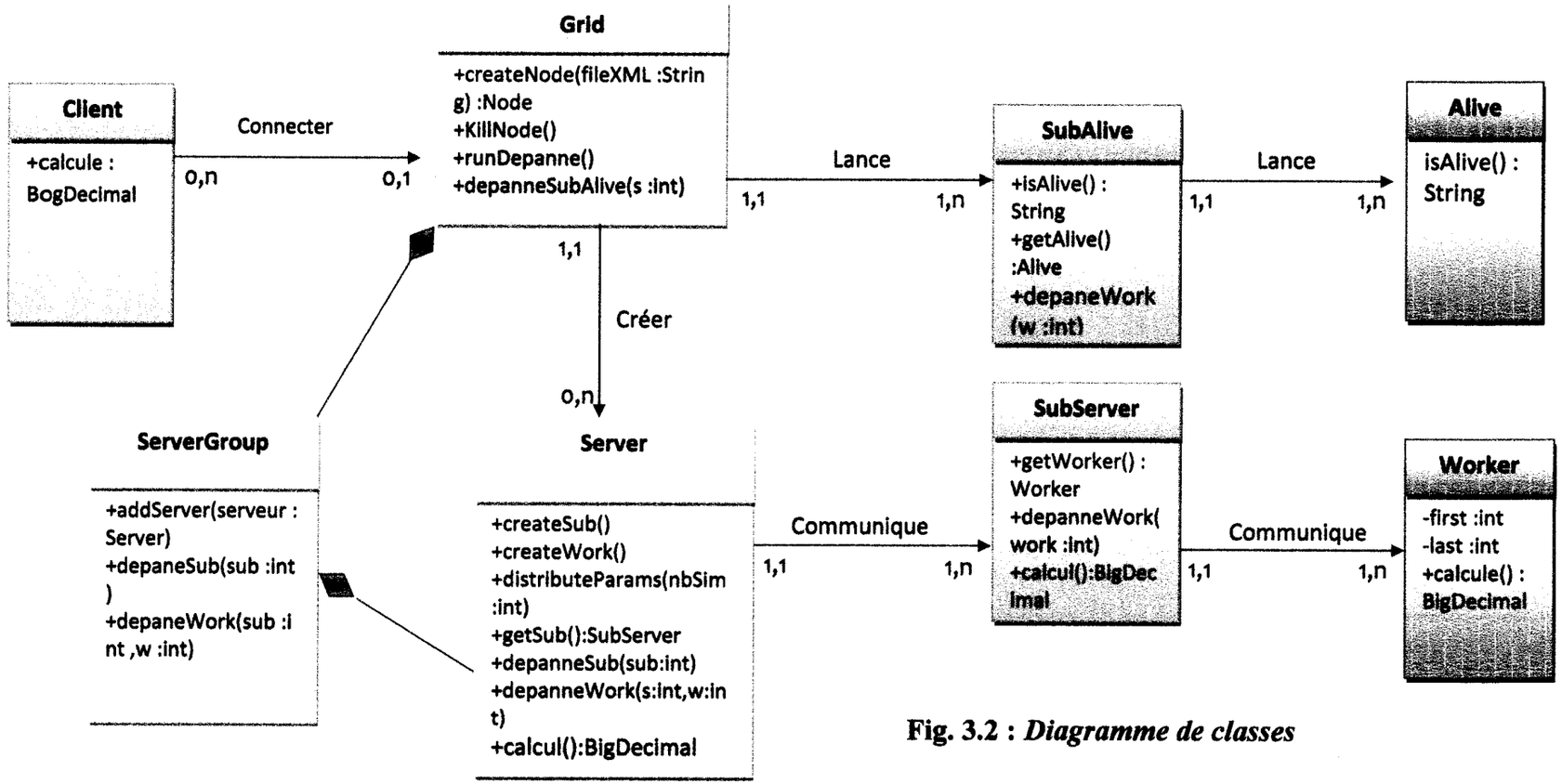

Fig. 3.2 : ***Diagramme de classes***

3.3.5 Information :

Nous avons réalisé ces diagrammes de façon qu'ils soient bien compréhensibles par le lecteur. Pour cela on a utilisé des noms assez significatifs pour les classes, les attributs et les méthodes ne portent pas réellement les mêmes noms dans le code source de notre application.

3.3.6 Conclusion :

La réalisation de ces diagrammes nous a beaucoup aidés dans le cycle de vie du développement de notre application. Cette modélisation nous a permis de réduire la complexité du projet ainsi qu'une meilleure conduite de ce dernier.

Chapitre 4 :

Architecture logicielle du projet

4.1 Principaux concepts de l'architecture logicielle "UnivGrid"

Le choix d'une architecture logicielle de grille est un point important dans la phase de conception, le plus souvent l'efficacité du système dépendra de l'architecture prise en conte. Dans notre cas, nous avons choisi une architecture dynamique qui s'adapte facilement avec le milieu extérieur pour résister à certain facteurs qui influence sur l'efficacité de notre système.

L'architecture ***UnivGrid*** que nous avons développée peut se déployer rapidement sur un cluster ou une grille multi-sites.

Les communications peuvent être réalisées par envoi de messages grâce à ProActive. Ce paradigme est mis à la disposition des utilisateurs pour leur permettre d'utiliser les ressources de la grille. Cette architecture est aussi générique dans le sens où elle permette à un utilisateur d'y inclure ses propres classes de calcul.

UnivGrid est composé de deux sous-systèmes :
Le premier sous-système à pour objectif de gérer continument le mécanisme de détraction et réparation de pannes.
Le deuxième sous-système s'occupe de la partie trainement afin de satisfaire les besoins du client.

4.2 Description détaillée de l'architecture logicielle "UnivGrid" :

Le système est hiérarchique : un Server répartit les calculs sur plusieurs **SubServers** qui eux-mêmes distribuent ces calculs sur leurs groupes de **Workers**. Ainsi, lorsque le nombre de **Workers** est très important (plusieurs centaines de machines), on peut augmenter le nombre de **SubServers** et diminuer la taille de chaque groupe de **Workers**. Cette stratégie permet d'éviter un goulot d'étranglement au niveau du Server, car le système ralentirait si le Server devait recevoir et traiter seul beaucoup de résultats venant directement de tous les **Workers**.

Le client envoie ses requêtes au Server en utilisant par exemple des *Sockets* (protocole TCP). Ce dernier va contacter les **SubServers** pour le lancement des calculs sur les **Workers** afin de satisfaire la demande du client. Ces mécanismes sont typiques des architectures 3 tiers. A l'opposé, la communication entre le **Server**, les **SubServers** et les **Workers** s'effectue principalement à travers les mécanismes de ProActive.
Le plus souvent un appel de méthode est lancé sur un *groupe* d'objets actifs composé d'un ensemble de **Workers** ou de **SubServers**. Ces mécanismes de communication de ProActive sont eux-mêmes implantés en RMI.
Notre architecture logicielle utilise intensivement les groupes d'objets actifs de ProActive.

Enfin, les "PC de réserve", incorporés dans l'architecture logicielle en fonction des besoins (récupération de panne), interviennent dans les mécanismes de tolérance aux pannes de ***UnivGrid*** (voir figure 4.1). On peut prévoir un nombre de machines de réserve correspondant à 2 ou 3 pourcents de l'ensemble. En général les services informatiques stockent un certain nombre de machines pour des remplacements d'urgence.

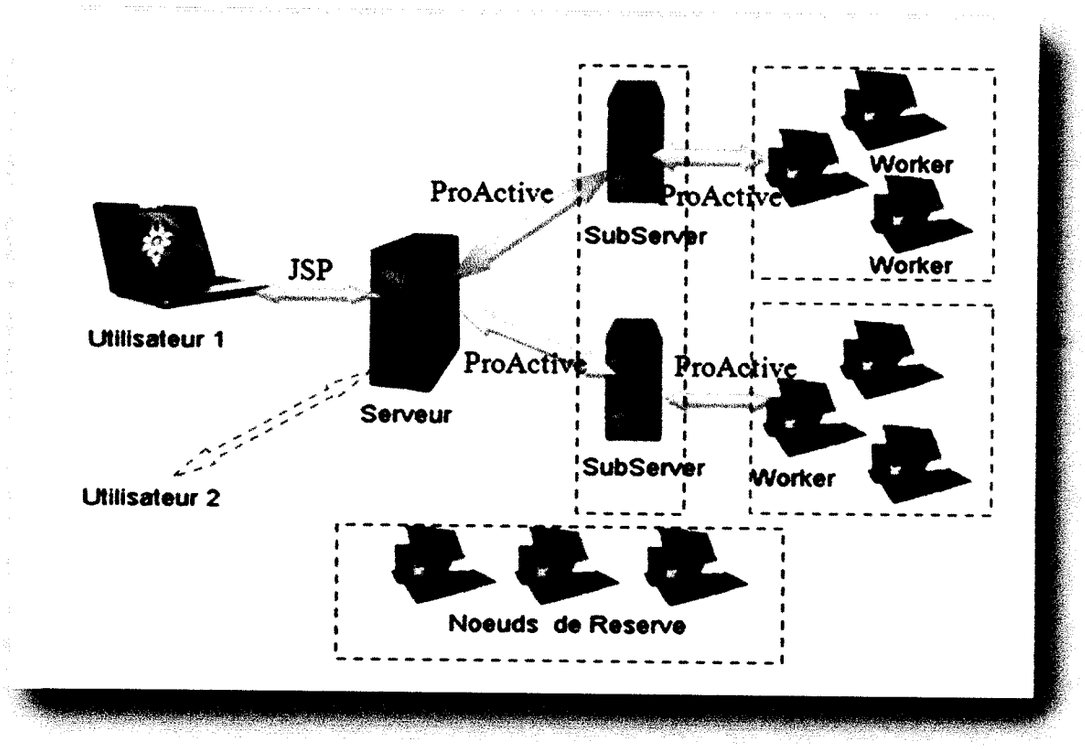

Fig. 4.1 L'Architecture UnivGrid

4.3 Mécanismes de déploiement de "UnivGrid :

Nous avons développé une interface graphique (voir figure 4.2) pour le Server d'UnivGrid qui contient un encadré "*Messages*" qui affiche les informations données par la partie système de l'architecture. L'interface affiche des informations nous permettant de connaître l'état d'***UnivGrid*** :

1-le nombre de nœuds déployés.
2-le nombre de machines de réserve
3-le nombre de Workers
4-le nombre de machines de réserve restantes suite à une panne.
5-les temps de calculs.

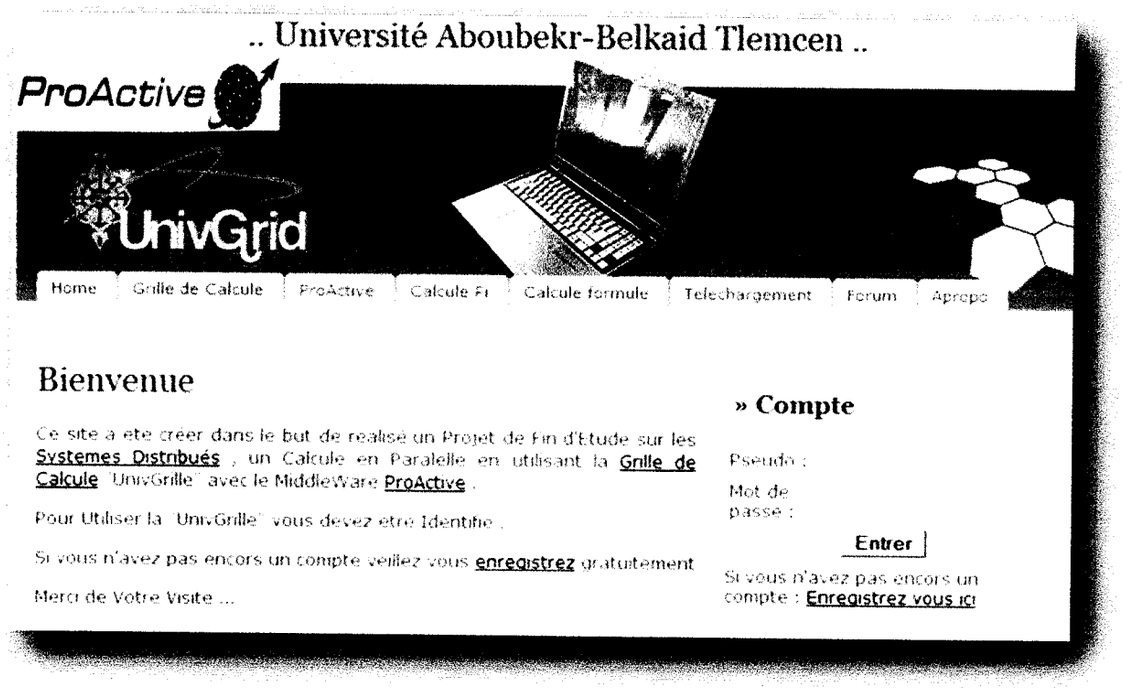

Fig. 4.2 Interface graphique

Un utilisateur peut à tout moment détruire tous les objets dont il lui sont réservés, on parle dans ce cas des SubServers et les Workers . Ensuite on peut à nouveau déployer UnivGrid avec un nombre de simulations identique ou différent.

Pour permettre à l'architecture ProActive de se déployer, on va utiliser un fichier *XML* de déploiement. Ce fichier *XML* est assez simple à créer, le chapitre 2 analyse un exemple en détail. Grâce à ce fichier *XML*, des nœuds sont créés sur toutes les machines du cluster ou de la grille que veut utiliser l'utilisateur. Ensuite, les SubServers et les Workers sont créés sur chacun des nœuds.

Le nombre total de SubServers et de Workers doit être inférieur au nombre de nœuds créés, les nœuds ne contenant pas d'objets sont gardés pour compenser les pannes (machines de réserve). L'encadré "*Messages*" de l'interface de la figure 4.2 nous indique tout ces informations .Le fichier *XML* de déploiement doit préciser quel protocole de connexion est utilisé pour joindre les machines : rlogin, rsh ou ssh.

4.4 Mécanismes d'équilibrage de charge :

4.4.1 Stratégie au niveau applicatif :

L'équilibrage de charge est un critère important dans l'architecture **UnivGrid** : Pour remédier à ce problème, on a créé une fonction qui s'en charge de tout, en ce basant sur une heuristique développée sur de nombreux tests réalisés auparavant.

Les calculs consistent donc à planifier un grand nombre de simulations, puis à distribuer ces simulations sur les Workers. L'application sera adaptée à une architecture homogène ou hétérogène (machines de puissance identique ou différente) en fonction des mécanismes d'équilibrage de charge.

La structure hiérarchique de l'architecture (Server, SubServers et Workers) permet d'effectuer un équilibrage de charge entre le Server et les SubServers, et entre chaque SubServer et son groupe de Workers. Dans chaque cas l'équilibrage de charge peut être statique ou dynamique (voir figure ci dessous).
Nous avons donc deux possibilités d'équilibrage de charge entre le **Server** et ses **SubServers** :

a) Un équilibrage de charge statique entre le Server et ses **SubServers** : le Server distribue le nombre total de simulations (N) à effectuer en le divisant par le nombre de **SubServers** (P), ainsi chaque **SubServer** réalise N/P simulations (répartition idéale avec des machines homogènes).

b) Un équilibrage de charge dynamique entre le **Server** et ses **SubServers** : le Server gère une réserve de tâches élémentaires, ces tâches sont distribuées sur les machines les plus performants de la grille.

De même on a deux possibilités d'équilibrage de charge entre un **SubServer** et ses **Workers**

a) Un équilibrage de charge statique entre un **SubServer** et ses **Workers** : le **SubServer** distribue le nombre de simulations attribuées par le Server en le divisant par le nombre de **Workers** (répartition toujours idéale sur une grille homogène).

b) Un équilibrage de charge dynamique entre un **SubServer** et ses **Workers** : chaque **Worker** va effectuer certain nombre de simulation, se nombre n'est pas fixe est vari selon la variation de la rapidité de la machine **Worker**.

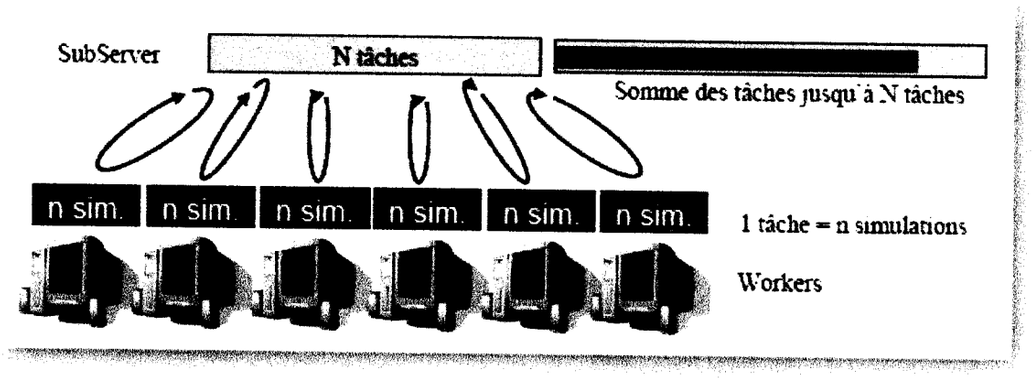

Fig. 4.3 Equilibrage de charges Statique [9]

Un équilibrage de charge statique entre le Server et ses **SubServers** et entre chaque **SubServer** et leur **Worker** est plus efficace lorsque les machines du cluster ou de la grille sont homogènes et lorsque les groupes sont homogènes. Mais un cluster ou une grille ne reste pas très longtemps homogène, car petit à petit certaines machines sont remplacées par de nouvelles plus puissantes, l'architecture devient alors hétérogène. Un équilibrage de charge dynamique ente le Server et ses SubServers et entre chaque SubServer et leur Worker est plus efficace pour une grille hétérogène, mais aussi pour une grille homogène car elle ne reste pas longtemps homogène.

L'utilisation de petites tâches de n simulations pour les Workers est indispensable pour minimiser les surcoûts lors des pannes mais demande plus de communication.

4.4.2 Exemple de mise en œuvre

Dans notre application ***UnivGrid***, nous avons utilisé la communication par RMI entre les SubServers et leurs Workers, pour implanter les calculs et évaluer les résultats.

Nous avons utilisé RMI et ProActive pour répartir les tâches et récupérer les résultats, nous avons implanté un équilibrage de charge statique entre un SubServer et son groupe de Workers pour le rendre plus tolérant aux pannes. Cette implantation rend l'application plus rapide sur un ensemble de PC homogènes et s'est avérée simple à réaliser en ProActive.

4.5 Mécanismes de tolérance aux pannes :

4.5.1 Stratégie à collaboration multi niveaux :

La figure 4.1 nous montre que l'application utilisant ***UnivGrid*** se divise en trois couches possédant chacune des mécanismes de tolérance aux pannes qui collaborent. Notre objectif est de limiter les pertes de temps pendant un calcul lorsque se produit une panne et de ne pas arrêter le reste du système pendant la réparation de cette panne. L'architecture doit pouvoir continuer à fonctionner normalement malgré des pannes ponctuelles, et en mode dégradé malgré des pannes répétées ou malgré la disparition d'un grand nombre de machines (par exemple suite à la coupure d'une partie du réseau). Les mécanismes applicatifs de tolérance aux pannes ont l'avantage de permettre une récupération sur panne plus précise que les mécanismes existant dans les middlewares, car ils sont adaptés à l'application : ils peuvent ne sauvegarder que le strict nécessaire, et ne relancer qu'un minimum de calculs. Cependant ils ne peuvent pas détecter ni réparer des pannes affectant le middleware.

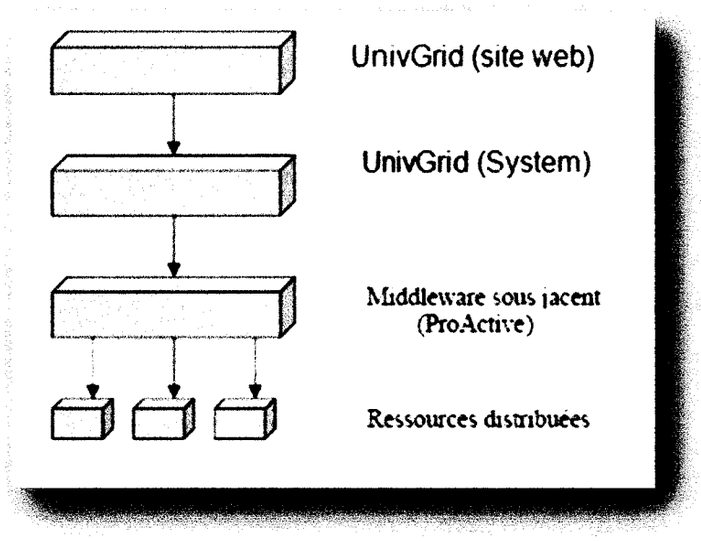

Fig. 4.4 Architecture multi niveaux

Nous avons donc choisi de conserver des PC de réserve, s'il en existe, pour maintenir en marche le système. Cette stratégie donne une grande fiabilité et une grande constance dans les outils de calculs, car en cas de panne leur entrée dans le système entraîne un surcoût limité et les temps d'exécution des calculs suivants redeviennent normaux.

4.5.2 Mécanismes au niveau du système "UnivGrid" :

La figure 4.6 nous présente les mécanismes développés au niveau d'UnivGrid qui fonctionne en utilisant ProActive. Ces mécanismes traitent essentiellement les pannes de machines plutôt que des pannes de processus. La "disparition" de machines est un problème fréquent sur les grilles, que nous traitons en incluant de manière contrôlée des PC de réserve, alors que les pannes de processus sont plus facilement détectables et réparables par les middlewares.

Une partie des mécanismes de tolérance aux pannes implantés au niveau de ***UnivGrid*** repose sur le déploiement d'objets actifs de ProActive , que l'on peut interroger à tout moment pour savoir si eux-mêmes et donc leurs machines hôtes sont "vivantes" : objets actifs "Alive".

Au moment du déploiement du système, chaque objet actif, SubServer et Worker, est créé sur une machine en compagnie d'un objet actif **Alive**. Le Server fait régulièrement un appel de méthode sur les objets **Alive** hébergés avec les SubServers. Si l'appel sur un objet **Alive** ne peut se faire, le Server considère que le PC contenant cet objet est en panne, et le SubServer associé est considéré perdu et devant être remplacé. S'il existe des machines de réserve, un nouveau SubServer est créé sur ce PC pour remplacer celui disparu. La même procédure est utilisée entre un SubServer et ses Workers. En ce qui concerne le temps d'attente entre chaque appel sur l'objet Alive, nous avons expérimentalement choisi une valeur de 100 millisecondes pour s'adapter a notre réseau. Cette valeur est utilisée par défaut par ***UnivGrid***, mais l'administrateur peut la modifier pour l'adapter aux caractéristiques de son réseau.

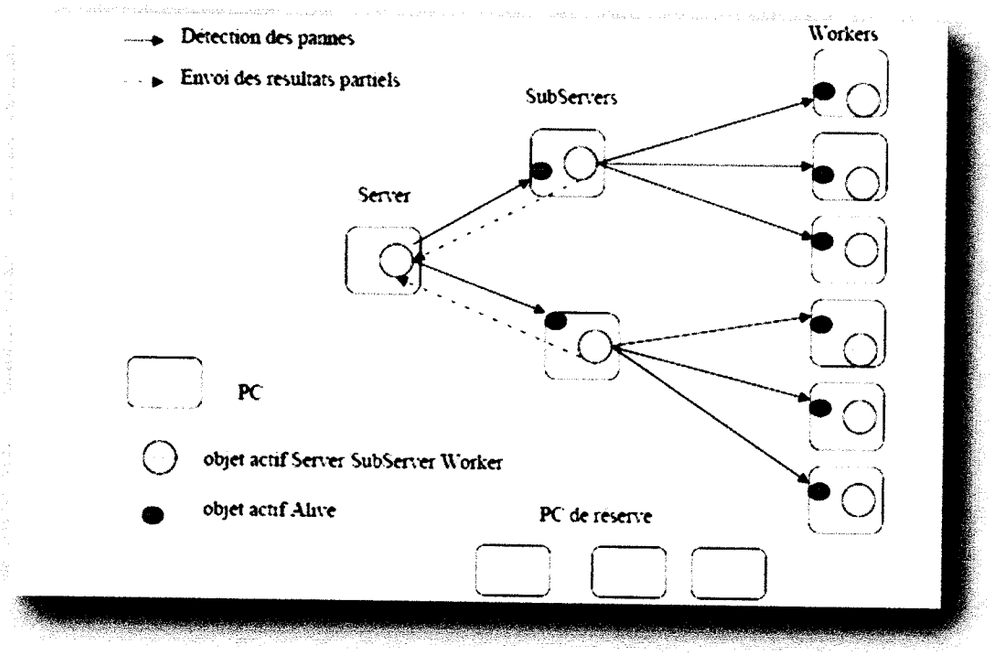

Figure 4. 5 Mécanismes de tolérance aux pannes

Si la panne d'un Worker se produit pendant une étape de calcul, un nouveau Worker est créé et peut effectuer les calculs, mais on perd le calcul commencé par l'ancien Worker depuis l'envoi d'une tâche par le SubServer. Donc plus les tâches seront petites, plus on réduit les temps de récupération sur erreur, mais plus on augmente les communications sur le réseau. L'administrateur fixera la taille des tâches en fonction des performances de son réseau.

Si un SubServer disparaît pendant un calcul, on relance le calcul sur le nouveau SubServer, Cependant nous souhaitons développer dans le futur un mécanisme de sauvegarde pour éviter de recommencer les calculs depuis le début, pour cela les SubServers doivent envoyer régulièrement au Server leurs résultats partiels et leur point d'avancement (*checkpointing*). Ainsi, le Server transmet au nouveau SubServer le nombre de tâches non réalisées par l'ancien SubServer et le dernier résultat partiel reçu du SubServer perdu, et le SubServer poursuit les calculs sans recommencer depuis le début. Si on augmente la fréquence du *checkpointing* on réduit les pertes de temps, mais on augmente les communications. L'administrateur fixera aussi cette fréquence en fonction des performances de son réseau.

Lorsqu'un Worker ou un SubServer est remplacé pendant des calculs, il faut fournir un nouveau Worker ou SubServer les données initiales nécessaires aux calculs.

Le Server sauvegarde une copie de la composition de tous les groupes de Workers, et si un Worker est remplacé dans un groupe, le SubServer gérant ce groupe envoie la copie du groupe transformé au Server. Lorsqu'un nouveau SubServer est créé, le Server lui transmet la copie à jour du groupe de Worker du SubServer en panne. Le nouveau SubServer peut alors relancer des calculs sur son groupe de Workers, mais en période de calcul il doit d'abord stopper tous ces Workers, car ceux-ci sont pour la plupart toujours en train d'exécuter une tâche demandée par le SubServer tombé en panne.

S'il n'y a plus de PC de réserve et qu'un Worker disparaît, le système continue à fonctionner mais avec un Worker en moins, donc les temps de calculs augmentent. Par contre, si un SubServer est perdu et s'il n'y a plus de machines de réserve, un nouveau SubServer est créé sur la machine hébergeant le Server, et les temps de calculs ne changent pas. Cependant plus des SubServers sont remplacés et plus le nombre de communications vers la machine hôte du Server augmente.

En fait la machine hébergeant le Server d'UnivGrid est considérée comme fiable et permet de détruire l'ensemble du système à tout moment. Une machine très robuste doit être utilisée comme serveur pour gérer une grille composée de PC ordinaires. Une architecture avec plusieurs Servers redondants se partageant les connexions des clients constituerait donc la future évolution d'UnivGrid.

4.5.3 Mécanismes au niveau applicatif

Pour que les mécanismes de tolérance aux pannes au niveau d'UnivGrid puissent fonctionner, il est nécessaire que l'utilisateur développe un minimum de méthodes pour pouvoir gérer ce mécanisme. Parmi ces mécanismes, on citera :

• fournir les données d'initialisation des calculs, dont la taille des tâches, lorsqu'il faut lancer un nouveau SubServer ou un nouveau Worker pendant les calculs suite à une panne.
• stocker le nombre de tâches à exécuter pour chaque SubServer et le fournir à un nouveau SubServer remplaçant un SubServer en panne.

Chapitre 5 :

Présentation de l'application

5.1 Introduction :

Notre application ***UnivGrid*** se présente sous forme d'un site web fait en JSP (Java Server Page , voir Annexe) qui est l'interface de communication entre les utilisateurs et la grille. Cette interface est riche en terme de fonctionnalités pour une bonne satisfaction à toute personne qui s'intéresse à ProActive (documentations , sources , et forum de discussion, etc.) ou bien une personne qui veut utiliser la grille. Tout cela est accessible à travers la page d'accueil qui se présente comme suite :

Fig. 5.1 Page d'index

5.2 Mode d'emploi d'UnivGrid :

Afin de pouvoir utiliser la grille de calcul il faut passer par les étapes suivantes :

5.2.1 Identification au system UnivGrid :

Le système UnivGrid est accessible à travers l'interface précédente, cela permet à un utilisateur distant d'accéder à la grille de calcul mais pour cela un pseudo et un mot de passe sont exigés.
Lors de l'identification du client le système doit d'abor se connecté à la base de donnée pour vérifier si le pseudo et le mot de passe existes, si oui le client pourra utiliser les ressources et lancer les calculs (pi et formule de Bessel), sinon UnivGrid fourni a l'utilisateur la possibilité de réessayer la connexion , ou de s'enregistrer(Voir Fig. 5.2).

Voici le code de la fonction qui permet cette vérification :

```
public static  boolean verif(String pseudo , String pwd){
      boolean test=false;
      try{
      Class.forName("org.gjt.mm.mysql.Driver");//charger le Driver
                  Connection
      c=DriverManager.getConnection("jdbc:MySQL://localhost/grid",
      "root","");  // se connecter a la base de donnée
      Statement s = c.createStatement();
      ResultSet r = s.executeQuery("SELECT * FROM client WHERE
pseudo='"+pseudo+"' AND pwd='"+pwd+"'" ); //executer la requette
      if (r.next()) test=true;//si le mdp et pseudo existe => true

      }catch(Exception e){e.printStackTrace();}
      return test;
}
```

Lors de son identification une ***Session***[1] sera créé pour le client afin de pouvoir basculer sur les différentes pages du site web en restant connecté (Voir la figure si dessous).

```
if(verif(pseudo,pwd)){
      session.setAttribute("pseudo",pseudo);//création session

}else  {
            //ici code du formulaire d'identification
      }
```

(1)Session : c'est un objet java qui permet de stocker des variables dans une durée bien définie

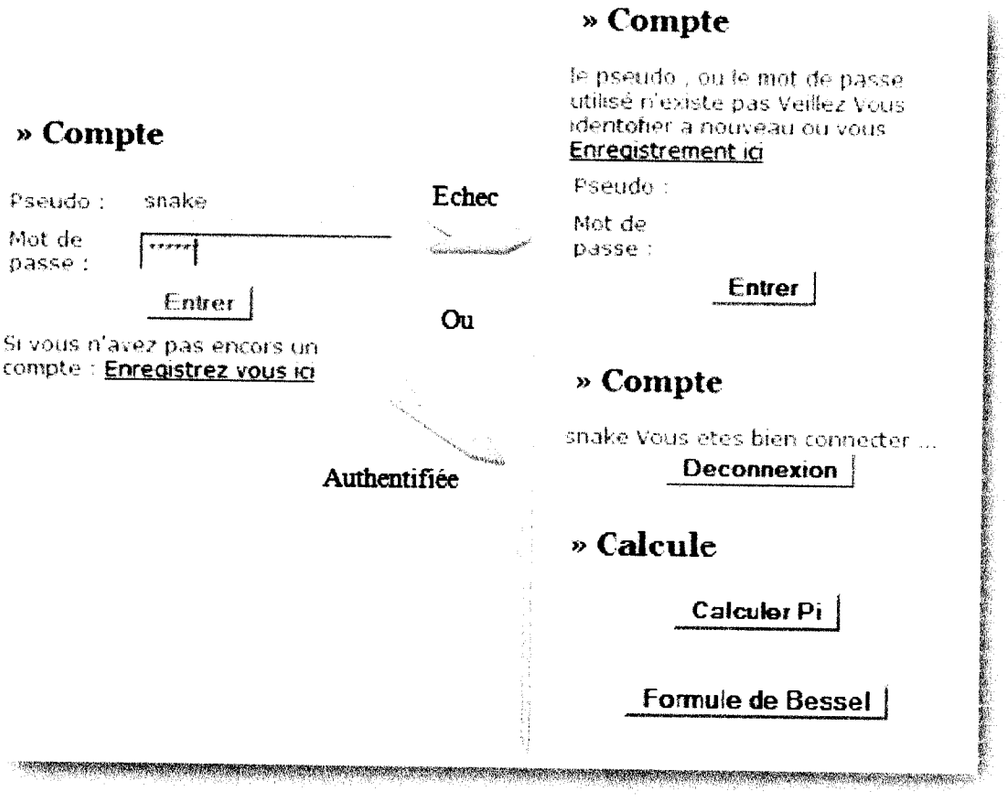

Fig. 5.2 Identification du client

5.2.2 Enregistrement du client :

Toute personne ne possédant pas un pseudo et un mot de passe peut s'inscrire a UnivGrid en se dirigeant sur la page d'enregistrement (voir la figure si dessous) , pour cela, l'utilisateur doit remplir ces information personnelle ainsi que le choix de son pseudo et mot de passe a travers un formulaire , après validation , ces information seront stocker dans une base de donnée pour une identification ultérieur (voir la figure si dessous).

Le pseudo doit être unique, c'est-à-dire qu'il est inutile d'enregistrer deux clients possédants le même pseudo, (On vérifie avec la base de données avant chaque enregistrement).

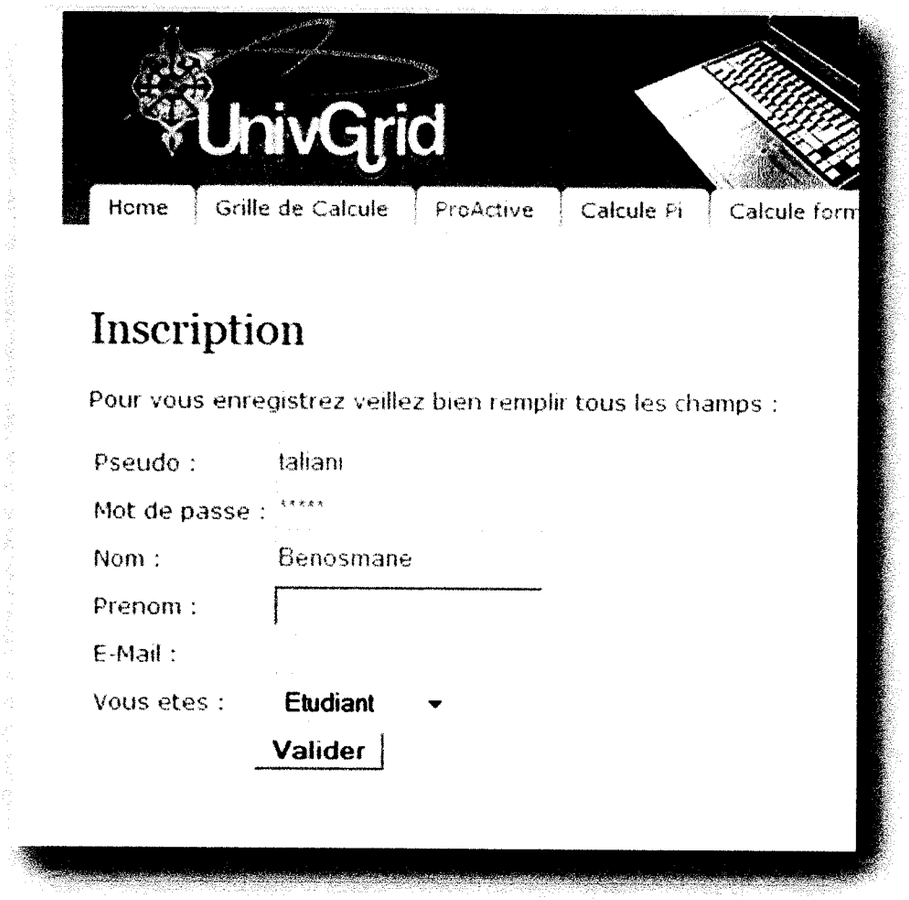

Fig. 5.3 Inscription d'un nouvel utilisateur

Voici le code de la fonction qui permet d'enregistrer le nouveau client :

```
try{
Class.forName("org.gjt.mm.mysql.Driver");
Connection c =DriverManager.
  getConnection("jdbc:MySQL://localhost/grid","root","");
Statement s = c.createStatement();
int r = s.executeUpdate("INSERT INTO
client(pseudo,pwd,nom,prenom,email,qui)VALUES('"+value[0]+"','"+value[1]+
"','"+value[2]+"','"+value[3]+"','"+value[4]+"','"+value[5]+"')");
}catch(Exception e){e.printStackTrace();}
out.println("Vous etes bien enregister"+value[0]+"<br /> Viellez vous
identifiez pour Utilisez nos ressources <br />");
```

5.2.3 Utilisation d'UnivGrid :

Une fois un client est identifié à UnivGrid il a l'accès à la ressource grille. L'acquisition des données initiales de calcul se fait via une page web (après l'identification du client) dans la quelle il peut insérer quelque paramètres qui sont nécessaires pour le lancement du traitement . (Voir la figure si dessous)

5.3 Comment est lancé le système UnivGrid ?

Le point d'entrée du système ***UnivGrid*** se situe au niveau du poste serveur c'est lui qui va s'en charger de construire toute l'architecture ,cette phase est critique et demande une bonne maitrise de quelque notions qui vont nous permettre d'atteindre ce but.

On va citer par la suite toutes les étapes nécessaires pour un bon lancement du système, ces étapes sont faites par la classe ***Grid*** une fois qu'elle est lancée.

La classe ***Grid*** n'est lancée qu'une seule fois et c'est au moment du démarrage du poste principal (server), elle possède des informations sur notre système ces informations sont ***public*** pour permettre l'accès via un autre poste.

5.3.1 Identification des postes de notre système :

Afin d'identifier les machines de notre système, on utilise un fichier XML (descripteur de déploiement).ce fichier représente les informations sur tous les postes de la grille, et il sera passé comme paramètre à des fonctions ProActive qui vont l'utiliser par la suite.

5.3.2 Création des nœuds :

A chaque poste de la grille un nœud va être créé sur lequel est hébergé les objets actifs grâce à la fonction **create_node()** dont le code est le suivant :

```
public void creat_node(String chemin ,String virtualnode_name)
{
try{
//déploiement des noeuds décrits dans le fichier Grid.xml
    ProActiveDescriptor pad =
ProActive.getProactiveDescriptor(chemin);
    pad.activateMappings();
    VirtualNode virtuelNoeudSalut =
pad.getVirtualNode(virtualnode_name);
    System.out.println(virtuelNoeudSalut.toString());
//  récupération des nœuds dans le  tableau tab_node
    tab_node= virtuelNoeudSalut.getNodes();
    System.out.println("récupération des noeud");
     }catch(Exception e)
     {
     e.printStackTrace() ;
     }
}
```

5.3.3 Classification des nœuds :

Suite à la création de nœuds, on doit désigner lesquelles vont représenter les SubServers et lesquelles vont représentés les Workers. Le choix du nombre de SubServers ainsi que le nombre de Workers vont se faire en se basant sur une heuristique acquise à travers de nombreux tests,mais dans notre cas on va se contenter d'un choix statique.

```
public  void set_node()
{
int i=0;
      tab_nb_worker=new int[nb_subserver];
      tab_node_subserver=new Node[nb_subserver];
      tab_node_worker=new Node[nb_subserver][];
for(i=0;i<nb_subserver;i++)
                              {

tab_node_subserver[i]=tab_node[indice];
                               indice++;
                              }
for(i=0;i<nb_subserver;i++)
                              {
                              tab_node_worker[i]=new
Node[tab_nb_worker[i]];
            for(int j=0;j<tab_nb_worker[i];j++)
                                              {
tab_node_worker[i][j]=tab_node[indice];
                                              indice++;
}}}
```

*** tab_nb_worker[i]:**est un vecteur qui represente le nombre de Workers pour le SubServer numero i ;

***tab_node_subserver:**est un vecteur qui porte les nœuds qui vont être considérés ensuite comme SubServer.

* ***tab_node_worker:*** est une matrice qui associe à chaque SubServer son tableau de Workers.

```
public static void get_host_information()
{
try{
      Grid.host_ip=InetAddress.getLocalHost().getHostAddress();
      Grid.host_name=InetAddress.getLocalHost().getHostName();
      System.out.println(Grid.host_ip);
      System.out.println(Grid.host_name);

}catch(Exception e){
      e.printStackTrace();
}

};
```

5.3.4 Récupération des informations du poste serveur :

Notre système possède une méthode qui lui permet de récupérer quelque informations sur le poste principal ainsi le système peut être lancé sur n'importe quelle machine.

A ce niveau, notre architecture est construite ainsi il est possible de créer des objets actifs sur chaque poste de notre grille.

5.3.5 Lancement du mécanisme de panne :

La classe ***Grid*** lors de son lancement va créer des objets Alive (Groupe d'Objet Actifs, voir chapitre 2) sur les différents postes de la grille de calcul ; ces derniers sont utilisés par le mécanisme de détection et réparation de panne.

Ce mécanisme tourne derrière un ***Thread*** pour assurer un contrôle continu et indépendant de la partie calcul.

Ce Thread va s'en charger de faire des appels de méthodes vers les différents postes de la grille ,ainsi il va assurer leur bon fonctionnement ; en cas où l'appel de la méthode se termine avec erreur , on procède à la réparation de la panne en remplaçant la machine tombant en panne par une machine de réserves .

5.3.6 Lancement du calcul :

Chaque requête reçu par le client crée une instance de l'objet Serveur, en lui passant les paramètres initiaux du calcul saisis préalablement (voir les figure 5.4 et 5.5) ,le paramètre le plus important est le nombre de simulation qui peut être représenté sous forme d' un intervalle ,cet intervalle va être divisé sur plusieurs petits intervalles, en premier temps le Server va créer un groupe de SubServer et plusieurs Groupes de Workers , chaque SubServer va gérer un groupe de Workers.
De cette manière chaque Worker va effectuer n simulations parmi N avec :

N : est le nombre de simulations totales.
n(i) : est le nombre de simulations élémentaires du Worker i.
Nb_worker : le nombre de Worker.

Quelque soit le nombre de Worker on a :

$$\sum_{i=1}^{i=Nb\_worker} n(i) = N$$

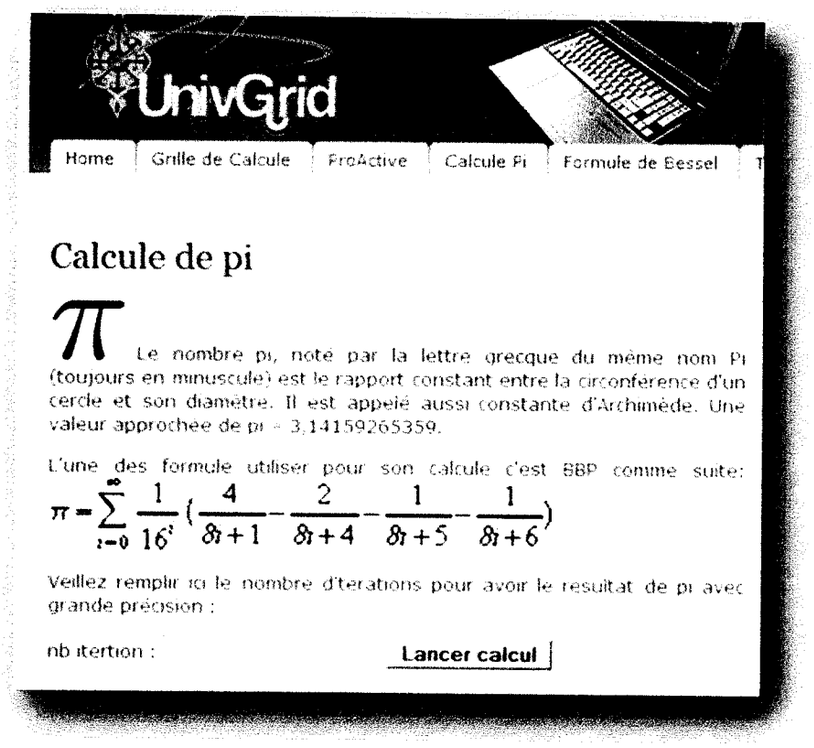

Fig. 5.4 Calcul de pi

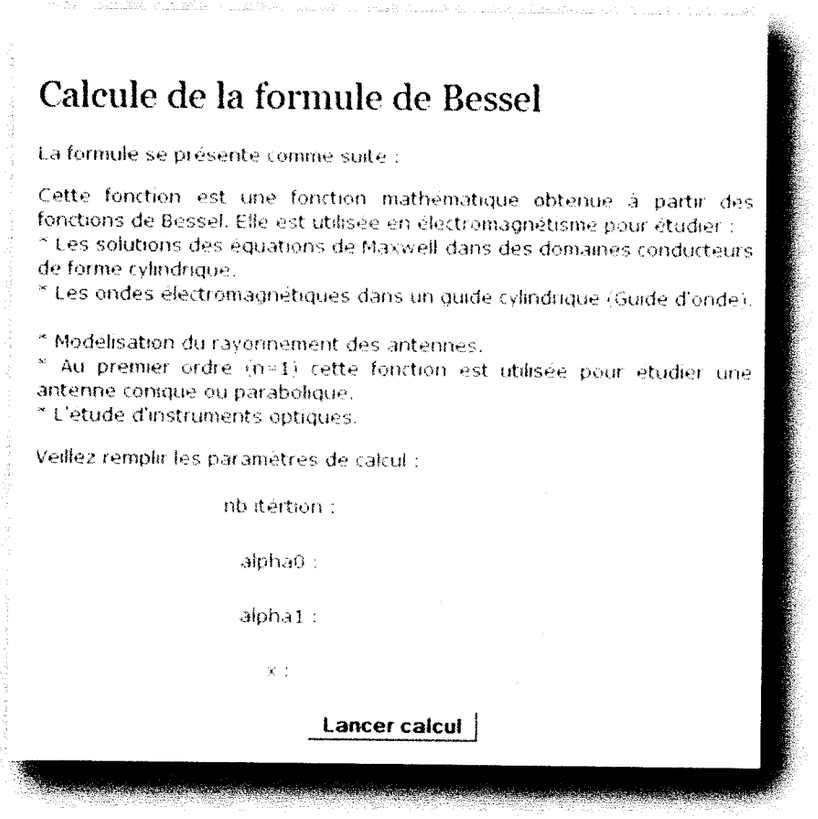

Fig. 5.5 Formule de Bessel

Voici le code de la fonction implémentée sur la classe Serveur qui devise les paramètres de calcul :

```
public  void distribute(int nb_Simulation) {
// recupérer la valeur de la jsp introdui par le client
      BigInteger nbSimulation=new BigInteger(nb_Simulation);
      int nbWorker=0;
//récupérer le nombre des workers totale
      for(int i=0;i<Grid.nbSubserver;i++)
nbWorker+=Grid.nbWorker[i];
      //nombre de simulation minimal de chaque worker
      BigInteger part;
      BigInteger[] finalPart =new BigInteger[nbWorker];
//nombre de simulation exacte exacte pour chque worker
      // remplir le tableau finalPart[]
      BigInteger diff;
      Integer nbWorker0= new Integer(nbWorker);
      BigInteger nbWorker1 = new
BigInteger(nbWorker0.toString());
      part=(nbSimulation.divide(nbWorker1));
      BigInteger diff0=nbWorker1.multiply(part);
      diff=nbSimulation.subtract(diff0);
      BigInteger one = new BigInteger("1");
      int diffInt = diff.intValue();
      for(int j=0;j<nbWorker;j++){
       if (diffInt>0){ finalPart[j]=part.add(one);diffInt--;}
       else finalPart[j]=part;
      }

      // Envoyer pour chaque worker son nombre de simulation a
effectuer

       Group gsubserver =ProActiveGroup.getGroup(gSubServer);
       BigInteger compt=new BigInteger("0");
       BigInteger first,last;     // debut et fin de sa
simulation
       int  kk=0;
       for(int i=0;i<gsubserver.size();i++){
            SubServer ss=(SubServer)gsubserver.get(i);
            Worker gworker =ss.getWorker();
            Group ggworker =ProActiveGroup.getGroup(gworker);

            for(int k=0;k<ggworker.size();k++){

                 first=compt;
                 last=(finalPart[kk].subtract(one)).add(first);
                 compt=last.add(one);


((Worker)ggworker.get(k)).attribute(first,last);

                 kk++;
}}}
```

5.3.7 Récupération des résultats :

Quand chaque Worker aura terminé son traitement il génère un bout de résultats, ce dernier va être transmis à son SubServer ; de cette manière chaque SubServer va avoir a la fin un ensemble de bouts de résultats, et il ne lui reste plus qu'à les transmettre à son Server, dès que le Server récupéra les résultats de ces SubServer il va générer le résultat final afin de pouvoir le transmettre au client ayant émis la requête, ce résultat va être disponible au niveau de la page JSP, ainsi on est resté fidèle aux contraintes de transparences.(voire Fig. 5.6)

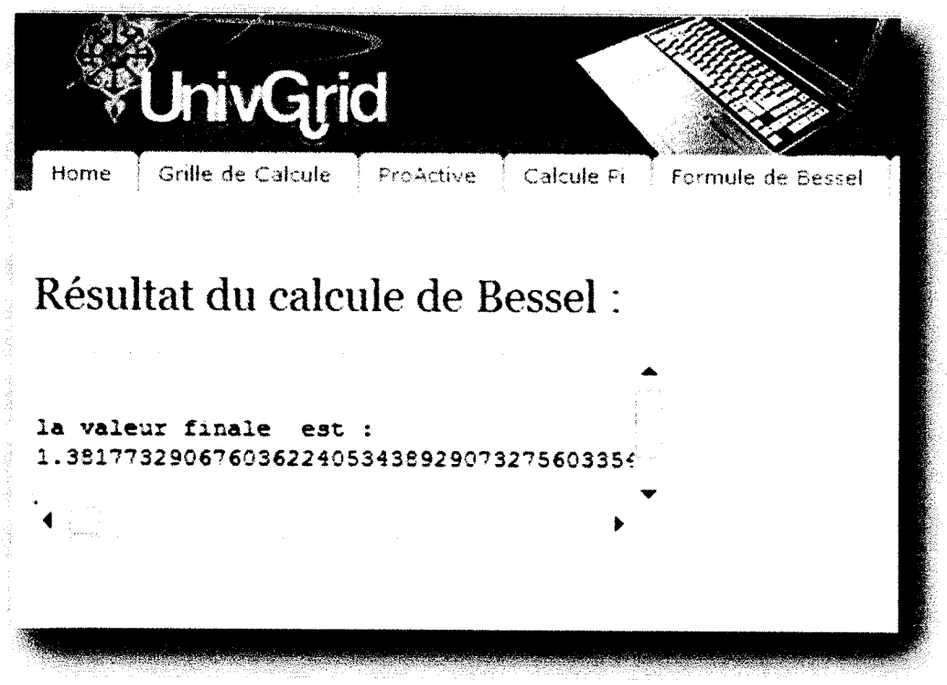

Fig. 5.6 Résultat de Bessel

5.3.7 Conclusion :

Notre architecture, *UnivGrid,* est performante le fait qu'elle donne pour tout type de traitement des résultats avec grande précision.
On a testé notre application sur une grille de deux PC qui a donné un résultat après 2mn, alors que le même traitement sur le monoposte donne un résultat après plus de 3 min.

Conclusion Générale

Pour nous ProActive était quelque chose de nouveau, c'est la raison pour laquelle on a mis beaucoup de temps pour s'habituer avec sa syntaxe dès que ce problème est résolu Toute la difficulté était dans l'adaptation de notre application avec l'architecture utilisée surtout qu'on a implémenté un mécanisme de tolérance aux pannes ; ce n'était pas évident de gérer les deux parties en même temps(partie Système, partie utilisateur). Tout ça sans oublier les problèmes rencontrés avec les protocoles de communication (SSH,RSH)qui ne sont pas implémentés sous WINDOWS alors on était dans l'obligation de travailler sous UNIX .

Nous avons beaucoup travaillé sur les précisions des résultats rendu par notre application, pour cela on a fait appelle a la classe BigDecimal car les autres types tel que les Integer, les Float sont révélés insuffisants (taille faible),mais il nous a fallu apporter quelques modifications de la classe BigDecimal car cette dernière n'était pas reifiable (voir titre 2.5).

Malgré tous les efforts que nous avons fait notre application n'est pas parfaite, certains critères doivent être améliorés .Comme perspective de ce travail, on citera :

Notre mécanisme de gestion de panne se résume à un Thread qui tourne continûment sur le poste serveur afin de récupérer l'état de notre système ainsi on peut facilement réagir en cas de panne .

L'administrateur de la grille fixera la durée de la période entre chaque deux contrôles successifs, deux critères dépendent de ce choix, à savoir l'efficacité de mécanisme de panne et la quantité de flux circulant sur la grille, mais le problème c'est que leurs dépendances sont inversement proportionnelles. Ce qui va être gagné en termes d'efficacité du mécanisme de panne va être perdu dans le sens d'augmentation de la charge sur le réseau.

Pour l'instant un équilibrage de charge statique est appliqué sur les différents Workers, l'influence de ce critère n'est pas aussi remarquable sur notre système puisqu'il est homogène. Mais pour plus d'efficacité on souhaite utiliser un équilibrage de charge dynamique qui consiste à allouer plus de traitements aux machines les plus rapides.

Il faut penser aussi à enrichi la grille en termes de ressources au fur et à mesure. Pour cela, on souhaite intégrer tout les machines des utilisateurs connectés à la grille pour participer au traitement.

Références

- **[1] Grid Computing** . Exposé Systèmes/Réseaux. [en ligne]

  http://www-igm.univ-mlv.fr/%7Edr/XPOSE2006/Jolly_Laskri/index.html

- **[2] HERA HOME** . Etude & Prototypage d'un GRID Service [en ligne]

  http://alexandre.richonnier.free.fr

- **[4] Equipe OASIS** . Communication de groupe typé pour objets répartis

  http://www-sop.inria.fr/oasis/personnel/Laurent.Baduel/publications

- **[5] ProActive** . INRIA Sophia Antipolis [En ligne]

  http://www-sop.inria.fr/oasis/proactive/

- **[6] F. Baude, L. Baduel , D. Caromel, A. Contes, F. Huet, M. Morel and R. Quilici.** Programming, Composing, Deploying for the Grid. In *"GRID COMPUTING: Software Environments and Tools"*, Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag, January 2006.

- **[7] Rabéa Ameur-Boulifa** . THÈSE : Génération de modèles comportementaux Dirigée par Eric Madelaine

  http://www-sop.inria.fr/dias/Theses/phd-153.php

- **[8] Arnaud CONTES** . THÈSE : une architecture de sécurité hiérarchique, adaptable et dynamique pour la grille.
  Dirigée par **Denis CAROMEL** et **Isabelle ATTALI**

- **[9] Sébastien Bezzine** . conception et implantation d'une grille de calculs de risques
  rapport de recherche en informatique - novembre 2006

  www.metz.supelec.fr/~ersidp/Publication/OnLineFiles/06-**Bezzine**.pdf

Références

- **[10] NICOLAS GAMA** , Projet de stage : Parallélisassions d'un solveur numérique d'équations électromagnétique – INRIA Sophia-Antipolis
  Dirigée par **Denis CAROMEL** .30 août 2003

  www.di.ens.fr/~pocchiol/RAPPORT-STAGE03/**gama**.pdf

- **[11] Antoine Dutot, Damien Olivier** , **Exposé : Objets distribués – Code mobile .Janvier 2005**

  http://www-lih.univ-lehavre.fr/~**dutot**/enseignement/CORBA/Cours/6Mobilite.pdf

- **[12] Fabrice Huet.** Objets Mobiles : conception d'un middleware et évaluation de la communication.
  PhD thesis, Université de Nice-Sophia Antipolis, 2002.

- **[13] Emmanuel Reuter**, THÈSE : Itinéraires pour l'administration système et réseau
  Équipe d'accueil : OASIS - INRIA Sophia Antipolis
  dirigée par Françoise BAUDE . 28 Mai 2004

- **Stéphane Genaud .** Grid Computing (exposé) [en ligne]

  http://icps.u-strasbg.fr/~genaud/

- **Stéphane Vialle.** *Parallélisme et Grid : Définitions*, 2006.

  http://www.metz.supelec.fr/~vialle/course/IIC-PG/index.htm

- **Grid'5000** : plateforme de recherche sur les grilles répartie sur le territoire français, 2006.

  https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home

Références

- **D.CAROMEL, W.KLAUSER, J.VAYSSIERE** : Twards Seamless Computing and MetaComputing in Java

  http://www.inria.fr/oasis/Denis.Caromel/ps/javallCPE.ps

  http://www.inria.fr/oasis/Denis.Caromel/ps/ic2dColor.ps

- **L.BADUAL, F.BAUDE, D.CAROMEL** : Efficient, Flexible and Typed Group Communication for java

  http://www.inria.fr/oasis/caromel/ps/GroupCommunicationforJava.pdf

- **Penser en Java** . Bruce Eckel [en ligne]

  http://penserenjava.free.fr

- **Développez** . Club d'entraide des développeurs francophones. [en ligne]

  http://www.developpez.com

- **L'encyclopédie infini** . Documentations java .[en ligne]

  http://www.infini-fr.com

Annexes

Annexe A : JDBC (Java DataBase Connectivity)

A.1 Introduction :

On a pu estimer que la moitié du développement de programmes implique des opérations client/serveur. Une des grandes promesses tenues par Java fut sa capacité à construire des applications de base de données client/serveur indépendantes de la plate-forme. C'est ce qui est réalisé avec Java DataBase Connectivity (JDBC).

A.2 Architecture client-serveur 2/tiers :

Dans une architecture client-serveur 2/tiers, un programme client accède directement à une base de données sur une machine distante (le serveur) pour échanger des informations, via des commandes SQL JDBC automatiquement traduite dans le langage de requête propre au SGBD.
Le principal avantage de ce type d'architecture est qu'en cas de changement de SGBD, il n'y a qu'à mettre à jour ou changer le driver JDBC du coté client. Cependant, pour une grande diffusion du client, cette architecture devient problématique, car une telle modification nécessite la mise à jour de chaque client.

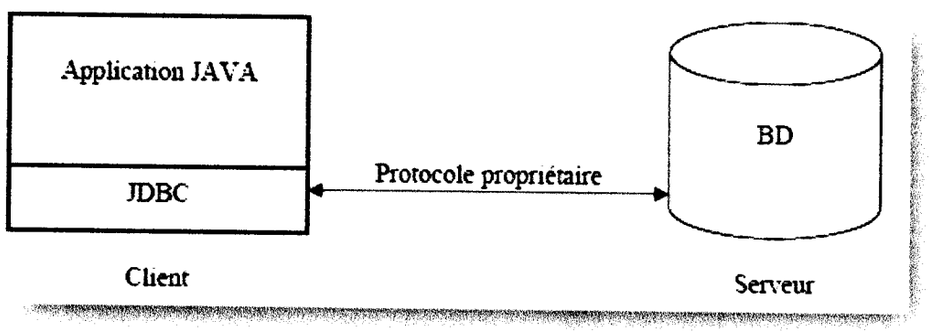

Fig. 1.1 Architecture client-serveur

A.3 Structure générale :

Pour effectuer un traitement avec une base de données, il faut :

1- Charger un pilote en mémo,

2- Etablir une connexion avec la base de données,

3- Récupérer les informations relatives à la connexion,

4- Exécuter des requêtes SQL et/ou des procédures stockées,

5- Récupérer les informations renvoyées par la base de données (si nécessaire),

6- Fermer la connexion.

A.4 Bibliothèques nécessaires :

Pour instancier les Objets nécessaires au dialogue avec une base de données, il faut importer les bibliothèques suivantes :

- **java.sql.*;**
- **sun.jdbc.odbc.*;** (pour inclure le pont JDBC-ODBC)

A.5 Charger un pilote en mémoire :

Avant l'utilisation d'un driver il faut le chargé en mémoire : ceci est fait en utilisant la méthode Class.forName("**org.gjt.mm.mysql.Driver**").

A.6 Différents types de pilotes :

1-Type 1 (JDBC-ODBC bridge) : le pont JDBC-ODBC qui s'utilise avec ODBC et un pilote ODBC spécifique pour la base à accéder. Cette solution fonctionne très bien sous Windows. C'est la solution idéale pour des développements avec exécution sous Windows d'une application locale. Cette solution « simple » pour le développement possède plusieurs inconvénients :

1- La multiplication du nombre de couches rend complexe l'architecture (bien que transparentes pour le développeur) et détériore les performances,

2- Lors du déploiement, ODBC et son pilote doivent être installé sur tous les postes où l'application va fonctionner,

3- La partie native (ODBC et son pilote) rend l'application moins portable et dépendant d'une plateforme.

2-Type 2 : un pilote écrit en java appelle l'API native de la base de données.

Ce type de pilote convertit les ordres JDBC pour appeler directement les APIs de la base de données. Il est de ce fait nécessaire de fournir au client l'API native de la base de données. Elles sont généralement en C ou en C++.

3-Type 3 : un pilote écrit en Java utilise un protocole réseau spécifique pour dialoguer avec un serveur intermédiaire.

Ce type de pilote utilise un protocole réseau propriétaire spécifique à une base de données. Un serveur dédié reçoit les messages par ce protocole et dialogue directement avec la base de données. Ce type de driver peut être facilement utilisé par une applet, mais dans ce cas le serveur intermédiaire doit obligatoirement être installé sur la machine contenant le serveur Web.

4-Type 4 : un pilote Java natif.

Ce type de pilote, écrit en java, appelle directement le SGBD par le réseau. Ils sont fournis par l'éditeur de la base de données. Ce type de driver est la solution idéale, tant au niveau de la simplicité que des performances et du déploiement.

A.7 Etablir une connexion :

La connexion à une base de données se fait par le biais de l'instanciation d'un objet de l'interface **Connection**. Elle représente une session de travail avec une base de données. L'interface **Connection** utilise les méthodes ***getConnection(...)*** de la classe **DriverManager** pour établir la connexion avec la base de données. Pour cela on passe l'url de la base de données en paramètre à la méthode.
Les méthodes ***getConnection(...)*** peuvent lever une exception de la classe **java.sql.SQLException**

```
try {
        Connection connect = DriverManager.getConnection(url);
    }catch (SQLException e){
                                                    ...
                                                }
```

A.8 Envoyer une requête :

- les **requêtes** sont représentées par la classe Statement ;
- les modifications de la base sont effectuées par la méthode Statement.executeUpdate(String)
- les requêtes sont effectuées par la méthode Statement.executeQuery(String);
- Le résultat d'une requête « Query » est un **ResultSet**

```
// On cree un canal de communication
            Statement st= connect.createStatement();
// On envoie une requête
ResultSet res= st.executeQuery("requette SQL");
// Tant qu'il y a des lignes dans le résultat..
while (res.next()) {
// on lit les valeurs des champs
System.out.println("col 1 = " + rs.getString("Nom"));
}
res.close();
st.close();
```

Note :
– On peut avoir plusieurs requêtes ouvertes sur la même connexion ;
– Il n'est possible d'accéder à un champ qu'une fois et une seule ;
– Il est nécessaire de fermer (close) les **Statement** et les **ResultSet**.

A.9 Conclusion :

L'utilisation des bases de données en Java, même si cela peut paraître ardu au premier abord est une gestion intéressante car indépendante du SGBD. Cela permet de faire du code réutilisable et portable sur n'importe quelle SGBD.

Annexe B : Les JSP (Java Server Page)

B.1 Introduction :

Les **JSP** (*Java Server Pages*) sont un standard permettant de développer des applications Web interactives, c'est-à-dire dont le contenu est dynamique. C'est-à-dire qu'une page web JSP (repérable par l'extension *.jsp*) aura un contenu pouvant être différent selon certains paramètres (des informations stockées dans une base de données, les préférences de l'utilisateur,...) tandis que page web "classique" (dont l'extension est *.htm* ou *.html*) affichera continuellement la même information.

Il s'agit en réalité d'un langage de script puissant (un langage interprété) exécuté du côté du serveur (au même titre que les scripts CGI,PHP,ASP,...)

Les JSP sont intégrables au sein d'une page Web en HTML à l'aide de balises spéciales permettant au serveur Web de savoir que le code compris à l'intérieur de ces balises doit être interprété afin de renvoyer du code HTML au navigateur du client.

B.2 Votre première JSP :

Afin de comprendre comment marche une page JSP, nous allons commencer notre étude par un petit exemple simple. Cette page JSP va afficher six titres de niveaux différents via les tags HTML *<H1>* à *<H6>*. Mais attention, ces tags ne vont pas explicitement être tapés, mais au contraire, générés via une boucle (voir le code si dessous).

```
<%@ page language="Java" import="java.util.*" %>
<HTML>
    <HEAD>
        <TITLE>Mapage_jsp</TITLE>
    </HEAD>
    <BODY>
       <H1 Align="center"> il est : <%= new Date() %></H1>

       <% for(int i=1; i<=6; i++) { %>
            <H<%= i %> align="center">Titre de taille <%= i %>
</H<%= i %>>
       <% } %>
       </BODY>
</HTML>
```

B.3 Aspects syntaxiques élémentaires :

Une page JSP se présente donc un peu comme une page HTML. Souvent, il y a plus de code HTML que de code Java. Mais des constructions nouvelles apparaissent. On peut facilement les reconnaître : elles commencent par les deux caractères <% et se terminent par %>. Ainsi, ce premier exemple (voir quelques lignes plus bas) commence par une construction indiquant quel est le langage à utiliser pour traiter la page ainsi qu'une demande d'utilisation de package. Cette première ligne est plus précisément une déclarative (elle commence par <%@).

D'autres bouts de code Java sont insérés dans la page, afin d'opérer la boucle sur le six niveaux de titres. Noter aussi que la date et l'heure exacte sont insérer dans la page via la construction <%=. En fait cette construction revient à taper <% *out.println(new Date());* %>, mais elle est plus concise. En conséquence, on est bien d'accord qu'il ne faudra jamais utiliser un point-virgule terminal dans ce type d'expressions.

B.4 Les expressions :

Les expressions JSP sont, comme leur nom l'indique, des expressions Java qui vont être évaluées à l'intérieur d'un appel de méthode *print* (voir l'exemple précédent). Une expression commence par les caractères <%= et se termine par les caractères %>. Comme l'expression est placée dans un appel de méthode, il est interdit de terminer l'expression via un point-virgule. Sans quoi une erreur de compilation vous sera retournée lors de la première invocation de votre JSP. Revoici un petit exemple d'expression JSP.

```
Il est : <%= new Date() %>
```

B.5 Les déclarations :

Dans certains cas, un peu complexe, il est nécessaire d'ajouter des méthodes et des attributs à la servlet qui va être générée (en dehors de la méthode de service). Une construction JSP particulière permet de répondre à ces besoins. Elle commence par les caractères <%! et se termine, vous vous en doutez, par les caractères %>. Voici un petit exemple d'utilisation.

```
<%@ page language="java" %>
<HTML>
   <HEAD>
        <TITLE>Exemple d'utilisation de déclarations JSP</TITLE>
        <%! private int userCounter = 0; %>
    </HEAD>
    <BODY>
        <H1 align="center">Exemple d'utilisation de déclarations
JSP</H1>

        <P>Vous êtes le <%= ++userCounter %><SUP>ième</SUP> client
du site</P>
    </BODY>
<HTML>
```

B.6 Les directives

Une directive permet de spécifier des informations qui vont servir à configurer et à influer sur le code de la servlet générée. Ce type de construction se repère facilement étant donné qu'une directive commence par les trois caractères <%@. Notons principalement deux directives : <%@ *page* ... %> et <%@ *include* ... %>. Voyons de plus près quelques unes des possibilités qui vous sont offertes.

La directive <%@ page .. %> permet donc, notamment, de pouvoir spécifier des informations utiles pour la génération et la compilation de la servlet. En fait, cette directive accepte de nombreux paramètres. Le tableau suivant vous présente sommairement, les principaux paramètres.

```
<%@ page language="java" contentType="text/html
import="java.util.*,java.sql.*" %>
```

Cette ligne indique tout d'abord que le langage de scripting utilisé dans la page JSP est Java, on utilise du format texte et html , et q'on importe les packages java.util et java.sql.

B.7 Résumé :

Au terme de cette annexe, nous avons présenté la technologie JSP. Il en résulte qu'une JSP est transformée en une servlet, et déployée, tous cela implicitement par le conteneur de servlet que vous utilisez. Ce qui, d'un certain pour de vue, est sympathique comparé aux servlets.
Paradoxalement, alors que les servlets mettent l'accent sur le code Java, les JSP, elles, favorisent le code HTML. De nombreuses constructions permettent malgré tout d'influer précisément sur le code Java auto-généré.