LIN Kaishin
SENG Alain
Promotion 2008

Rapport de projet 2A

Interception système pour l'émulation d'applications

Correspondant école :
Phuong LE-HONG

Responsables Loria :
Lucas NUSSBAUM
Martin QUINSON

Introduction

Ce projet s'inscrit dans le cadre des projets de deuxième année du département Information et Systèmes de l'Ecole des Mines de Nancy. Ces derniers ont pour objectif d'initier à la gestion concrète d'un projet en groupe, mais également de permettre aux élèves d'approfondir leurs connaissances dans un domaine technique et scientifique qui ne serait éventuellement pas abordé en cours. Le projet se déroule sur l'année scolaire 2009-2010 au sein de l'équipe AlGorille du Loria, et est encadré par deux de ses membres M. Lucas NUSSBAUM et M. Martin QUINSON.

Table des matières

INTRODUCTION **2**

TABLE DES MATIERES **3**

I. GESTION DU PROJET **4**

CONTEXTE 4
ENJEUX 4
CAHIER DES CHARGES INITIAL 5
Objectifs *5*
Fonctions attendues *5*
Critères d'évaluation et d'acceptabilité *6*
Contraintes et modalités de réalisation *6*
Echéancier *6*

II. TRAVAUX REALISES **7**

DEVELOPPEMENT D'APPLICATIONS DISTRIBUEES 7
INTERCEPTION : LES AGENTS JAVA 8
Le contexte *8*
Les objectifs fonctionnels et techniques *9*
Les moyens disponibles *10*
ADAPTATION AVEC SIMGRID 11

III. DIFFICULTES RENCONTREES **13**

DIFFICULTES AU COURS DES REALISATIONS 13
DIFFICULTES SUR LA GESTION DE PROJET 13

CONCLUSION **15**

ANNEXES **16**

A. LES DIFFERENTES APPLICATIONS DISTRIBUEES DEVELOPPEES 16
La classe Message *16*
Intérêt du multithreading *16*
Ping-pong : le modèle du Client/Serveur *17*
Token Ring : structure en anneau *17*
All-to-All *17*
Scatter-Gather *17*
B. COMPLEMENTS TECHNIQUES SUR L'INTERCEPTEUR 18
Le Manifest et le fichier jar *18*
Le fichier Agent.java *18*
ASM : ClassRenamer.java *19*
Message.java et RealMessage.java *19*

I. Gestion du projet

Contexte

La recherche dans de nombreux secteurs nécessite une puissance de calcul croissante et, la mise en commun des ressources de plusieurs ordinateurs permet de répondre à ce besoin. Malheureusement, l'écriture d'applications destinées à fonctionner sur des systèmes distribués n'est pas simple. Les expérimentations jouent un rôle crucial dans cette conception car les modélisations théoriques qui pourraient décrire l'application sont, soit trop complexes pour être exploitables, soit trop simplistes pour être réellement utiles.

Pour les expérimentations, deux possibilités s'offrent alors aux programmeurs :

- soit, ils effectuent leurs tests sur une plate-forme, copie conforme de celle où sera installée l'application (achat de matériel en double)
- soit ils procèdent à des simulations.

Les inconvénients de la première solution sont de taille. Elle nécessite en effet l'écriture préalable du programme complet avant de pouvoir procéder aux expérimentations, ce qui rend ardue la comparaison entre les différentes solutions possibles et dont on voudrait justement évaluer les avantages ou inconvénients respectifs. De plus, cette solution pose des problèmes de reproductibilité : il est difficile d'isoler une plate-forme réelle pour procéder aux expérimentations *in situ* et les autres utilisateurs peuvent influer sur les résultats.

La simulation ne présente pas les problèmes énumérés ci-dessus mais nécessite l'implémentation d'un prototype destiné à fonctionner sur le simulateur, puis une réécriture de l'application destinée à fonctionner sur la plate-forme réelle (une fois le prototype validé). Ecrire l'application deux fois représente une surcharge de travail qui pourrait être évitée si on parvenait à effectuer des simulations directement à partir de l'application réelle. Au cours de l'implémentation du prototype, des écueils divers peuvent en outre surgir : situations de compétition (*race condition*), interblocages (*dead locks*), famines, absence d'horloge centralisée et donc asynchronisme…

Enjeux

Le projet s'insère donc dans un axe de recherche visant à permettre la conduite de simulations sans avoir à implémenter de prototype, d'où un gain de temps et de fiabilité. Par exemple, la vérification d'absence de divergence entre le modèle et l'application réelle n'aurait plus lieu d'être, éliminant une source potentielle d'erreurs.

Les tests d'applications déjà existantes s'en trouveraient également facilités puisque l'étape délicate de leur modélisation, pour laquelle il faut décortiquer le fonctionnement et donc comprendre exactement le travail d'un tiers, ne constituerait plus un écueil.

Cahier des charges initial

L'établissement du cahier des charges correspond à un jalon important de la gestion de projet que nous avons établi début novembre. En plus des objectifs et d'un descriptif des fonctions attendues, il comporte un échéancier qui s'est révélé au final être assez éloigné de la réalité. Nous y reviendrons dans la partie III consacrée aux problèmes rencontrés lors de la gestion de projet.

Objectifs

L'objectif est donc de permettre à des applications réelles de s'exécuter directement dans un simulateur. Pour ce faire, il est nécessaire de :

1. Etudier les moyens d'intercepter les actions d'une application
2. Implémenter des exemples d'applications distribuées
3. Implémenter une méthode d'interception qui redirige les interactions avec l'environnement vers le simulateur SimGrid

Fonctions attendues

Il existe différentes pistes pour intercepter les actions d'une application : l'appel *ptrace* du debugger gdb, le détecteur d'anomalie valgrind, l'éditeur de liens dynamiques (ldd) de linux, les agents utilisés dans Java. Dans le cadre du projet, on ne s'intéressera qu'à cette dernière piste.

Il est demandé aux étudiants qu'ils développent quelques exemples d'applications distribuées plus ou moins complexes en Java afin de disposer d'exemples à tester intéressants par la suite.

En plus de pouvoir procéder simplement aux interceptions, le développement de l'intercepteur doit prendre en compte la nécessité d'interagir avec le simulateur SimGrid. Les interceptions réalisées devront lui permettre de simuler le temps de calcul qu'il aurait fallu à une plate-forme réelle pour exécuter les instructions demandées.

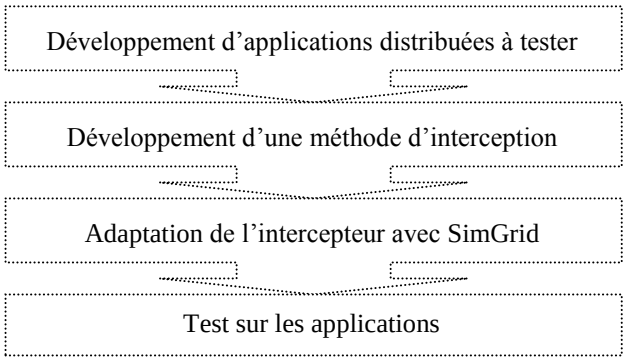


Fig.1 : Résumé des fonctions attendues

Critères d'évaluation et d'acceptabilité

A la fin du projet, il est donc attendu qu'un ensemble de petites applications distribuées en Java ait été implémenté. Celles-ci devront être fonctionnelles, documentées et avec un code le plus clair possible. Entre 4 et 6 applications différentes sont prévues.

L'intercepteur doit être fonctionnel et doté de la couche d'interaction avec SimGrid. Il sera validé en utilisant la collection d'applications distribuées évoquée ci-dessus.

Contraintes et modalités de réalisation

Les contraintes techniques sont définies par rapport aux connaissances des élèves travaillant sur le projet mais qui, bien sûr, évolueront au fil de leurs recherches personnelles.

Chaque lundi après-midi, un créneau horaire est dédié à l'avancement du projet. En plus des rencontres avec les encadrants du Loria qui pourront avoir lieu à cette occasion, un espace de travail employant SVN a été mis en place sur le site internet Assembla pour faciliter la communication au sein des membres du projet. D'autres réunions pourront être prévues en dehors de ces horaires selon les besoins et les disponibilités.

Echéancier

Du 16 novembre 2009 au 17 janvier 2010 (9 semaines)

Développement de la collection d'applications distribuées.

Du 18 janvier 2010 au 7 mars 2010 (7 semaines)

Développement de l'intercepteur.

Du 8 mars 2010 au 25 avril 2010 (7 semaines)

Réalisation de la couche permettant l'interaction avec le simulateur SimGrid.

Du 26 avril 2010 au 13 juin 2010 (7 semaines)

Test des applications distribuées sur SimGrid et validation de l'intercepteur.

Il s'agit bien sûr du calendrier initial prévu. Un diagramme de Gantt est disponible dans la partie III pour comparer l'avancement prévu et effectif du projet.

II. Travaux réalisés

Avant même de commencer à travailler à proprement parlé sur le projet, nous avons du nous l'approprier et saisir les enjeux expliqués dans la partie précédente. Ainsi, bien que l'échéancier du cahier des charges initial ne commence qu'à partir du 16 novembre, nous nous sommes entretenus avec nos tuteurs de laboratoire dès le début du mois d'octobre pour essayer de bien saisir tous les tenants et aboutissants.

Au cours de ces entretiens, nous avons également convenu d'utiliser un outil que nous ne connaissions pas jusqu'alors : SVN qui permet de gérer les différentes versions d'un code source (et de documents) au fur et à mesure des modifications apportées. SVN était couplé avec l'espace de travail partagé Assembla, accessible sur Internet, afin de pouvoir tenir M. QUINSON et M. NUSSBAUM informés de l'avancement du projet.

Une fois achevée cette première phase d'analyse, parfois fastidieuse mais néanmoins indispensable, nous avons rédigé le cahier des charges ci-dessus et avons commencé à réaliser les fonctions attendues.

Développement d'applications distribuées

Nous avions choisi de commencer par cette phase plutôt que par l'interception car il semblait plus simple et plus progressif au niveau de la difficulté de procéder ainsi. En effet, nous verrons qu'autant cette phase d'implémentation s'est relativement bien déroulée, autant la seconde phase s'est révélée plus ardue

. Le choix de développement des différentes applications distribuées s'est fait, sur les conseils de nos responsables de projet, vers celui des différents types de réseaux considérés comme classiques et qui permettront de **tester notre futur intercepteur avec des applications intéressantes et variées**. Un descriptif plus détaillé de chacune d'entre elles est disponible dans l'annexe A.

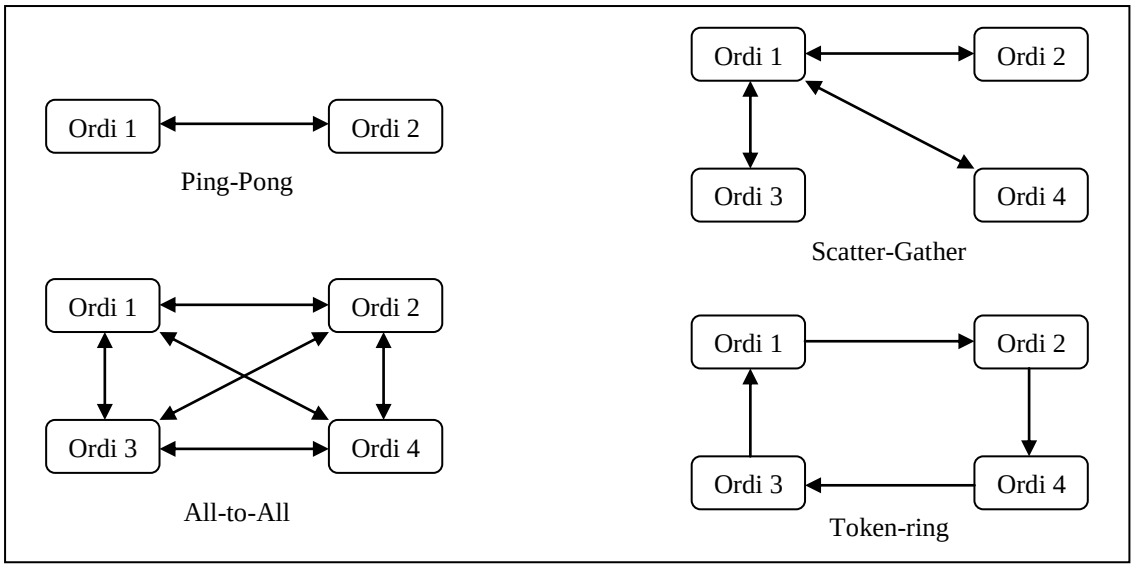


Fig.2 : Communications au sein des différents types d'applications développées

Nous codions en java à partir de l'IDE Eclipse. Grâce à cette phase du projet, nous avons appris à mieux maîtriser l'implémentation d'architectures clients/serveurs et à nous familiariser avec les notions de compilation en dehors d'Eclipse, chose à laquelle nous n'avions jusqu'alors pas été confronté dans le cadre de nos cours.

Du fait de notre inexpérience dans le domaine des applications distribuées, nous avons du consacrer un temps relativement conséquent à cette phase.

Interception : les Agents Java

Le contexte

Intercepter une application ressemble à l'espionnage d'une application : afin de connaître les temps de calcul, il faut d'abord savoir de quoi ces calculs sont constitués. De ce fait, il faudra que notre intercepteur sache fournir en temps réel les activités d'un programme. Toutefois, l'espion ne doit en aucun cas perturber le bon fonctionnement du programme espionné. Ce dernier doit s'exécuter indifféremment de l'intercepteur.

L'important n'est pas le résultat d'un calcul mais le temps de calcul. De ce fait, l'intercepteur se doit de modifier quelques données au cours de l'exécution de l'application. **Il doit donc anticiper et réagir face à une application sans perturber son exécution.**

L'idée est de modifier le *bytecode* d'une classe donnée lorsqu'elle est appelée, par un autre *bytecode* légèrement modifié par l'intercepteur pour accomplir les actions souhaitées. Lorsque la JVM charge une classe, elle « l'appelle ». Quand cette classe commence à être chargée, l'intercepteur intervient et la JVM charge une fausse classe fournie par l'intercepteur. Voici un schéma explicatif sur le fonctionnement de l'interception :

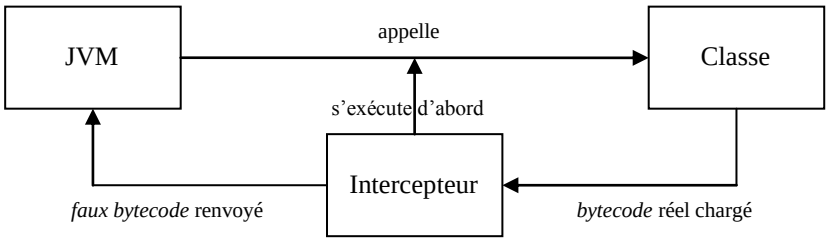


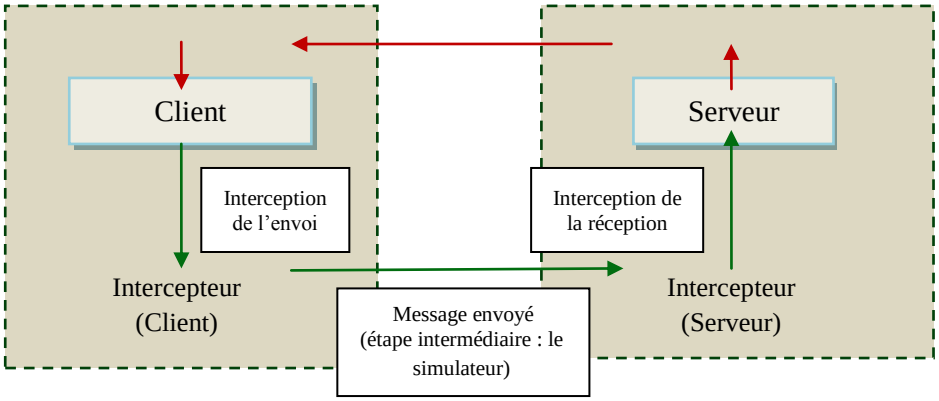
Fig.3 : Schéma de principe de l'interception.

Il existe plusieurs méthodes envisageables pour effectuer l'interception et qui sont citées dans le cahier des charges. Avant de le rédiger, nous avions toutefois convenu avec nos responsables de projet que nous ne nous consacrerions qu'aux agents java, introduit avec le JDK 1.5 car nous étions plus à même de travailler dans un environnement qui nous était déjà familier (ou du moins, moins inconnu que les autres possibilités) et pour plusieurs raisons exposées ultérieurement.

Les objectifs fonctionnels et techniques

L'intercepteur ne doit en aucun cas perturber le bon fonctionnement du programme espionné. Ce dernier doit s'exécuter indifféremment de l'intercepteur. De même, l'espion ne doit pas non plus ralentir significativement le programme principal. L'interception doit **être invisible aux yeux des applications espionnées**.

D'autre part, les informations recueillies seraient inutiles si elles n'étaient pas immédiatement traitées. Traiter une information, c'est la transformer afin « d'accélérer » l'expérience ou le calcul. On supprime ainsi les informations dont on n'a pas besoin, comme par exemple les résultats d'un calcul mais on conserve les informations essentielles à la simulation sur SimGrid comme le temps de calcul. Il n'est d'ailleurs pas nécessaire d'attendre le temps de calcul en entier : l'intérêt du monde virtuel est que l'on peut « avancer » le temps du référentiel du programme cible. L'intercepteur doit uniquement **fournir les données nécessaires à la simulation sur SimGrid**.



Supposons qu'on soit dans le cas où un client demande un calcul important :

1. Le message est envoyé par un client.
2. L'intercepteur capte le message et modifie le contenu.
3. Ce message modifié est envoyé normalement
4. C'est entre les deux intercepteurs qu'interviendra le simulateur SimGrid.
5. L'intercepteur serveur reçoit le message et le modifie au besoin. Le serveur comprend qu'il y a un message mais ne sait pas qu'il est modifié.
6. Si le calcul prend n secondes, temps déterminé par le simulateur, le serveur avance l'horloge virtuelle de n secondes sans calculer. Il envoie sa réponse.
7. La réponse subit le même processus jusqu'à arriver au client.
8. Ce dernier ne voit que l'avancé de son horloge virtuelle de n secondes.

En un laps de temps très court, nous avons pu voir combien de temps prend ce calcul. Le calcul du serveur dépend de la structure de l'application distribuée, rôle du simulateur, auquel il est rattaché.

Fig.4 : Résumé du déroulement d'un envoi et réception de message

En outre, ce projet est destiné à être distribué. Ainsi, au-delà des simples objectifs, se trouvent des contraintes techniques qui doivent faciliter l'usage des intercepteurs aux utilisateurs qui ne connaissent pas leur implémentation.

L'optique du projet est de créer des modules qui complètent un programme. Nous avons donc évité d'utiliser au maximum des sources externes, dans l'idée d'**assurer la portabilité de notre intercepteur**, d'où l'orientation de notre choix vers les agents java (cf. les moyens mis en œuvre). Le déploiement de l'intercepteur ne doit pas s'accompagner de multiples téléchargements ou installations diverses.

Notre seconde priorité est ensuite la robustesse : notre intercepteur doit **être le plus fiable possible quelles que soient les applications interceptées**. En effet, notre intercepteur travaille à un niveau où il est parfois difficile de saisir tout ce qui s'y passe. A ce niveau, la fiabilité n'est pas de 100%, il faut donc privilégier la méthode la plus sûre.

La troisième priorité concerne la précision de l'intercepteur. En effet, l'interception est un mécanisme qui peut se révéler complexe et peut ralentir le fonctionnement de l'application s'il n'est pas optimal. Optimiser l'interception, c'est **choisir les classes les plus pertinentes à intercepter**.

Les moyens disponibles

Fondamentalement, nous devions choisir une technologie nous permettant la modification de classes. Deux solutions principales se sont alors offertes :

- les décompilateurs tels que libASM
- les agents java

LibASM (ASM pour assembleur) est une librairie extrêmement puissante qui permet de décompiler une classe au vol, de la modifier puis de la recompiler. LibASM travaille alors sur les fichiers *.class*, issus de la compilation Java. Le principe de base de libASM est de « visiter » une classe : après avoir chargé une classe, écrite alors dans un langage qui ressemble à un langage assembleur, libASM peut lire ligne par ligne le code de cette classe et éventuellement la modifier. Cependant, la modification s'effectue aussi en langage pseudo-assembleur. Un fichier *.class* Java n'est pas réellement écrit en langage assembleur.

LibASM reste un outil très puissant mais si toute l'interception ne se base que sur cette libraire, cela nous imposerait une forte dépendance, contredisant notre objectif de portabilité. De plus, il s'agit d'un outil assez lourd à mettre en œuvre que nous avons souhaité éviter dans la mesure du possible.

Les agents java constituent une alternative intéressante. L'idée est alors d'avoir un programme prioritaire en ce sens qu'il doit s'exécuter impérativement avant les autres dans la machine virtuelle. Ce type de méthode existe, ce sont les *premains*, **s'exécutant avant toute autre classe,** qui sont des méthodes propres aux agents java.

De plus ils ont la **particularité de s'attaquer à un niveau relativement bas** du code : le chargement des classes. C'est un atout indéniable puisque notre intercepteur a intérêt à viser le dénominateur commun de chaque programme pour être le plus général possible.

L'intercepteur se base sur le **fonctionnement séquentiel de la JVM** (*Java Virtual Machine*). Chaque classe appelée par un programme doit préalablement être chargée avant son exécution et ces chargements sont discrétisés par la machine virtuelle qui les traite une à une : deux classes ne peuvent être chargées simultanément. Chaque classe peut donc être interceptée par un agent java.

	Avantages	**Inconvénients**
LibASM	Très puissant	Lourd à mettre en œuvre
Agents java	Portable (intégré à Java) Suffisant pour notre objectif	

Fig.5 : Comparatif des moyens d'interception

Pour davantage de détails sur l'implémentation de l'intercepteur avec les agents java, nous invitons le lecteur à se référer à l'annexe B en fin de rapport.

Adaptation avec SimGrid

Une fois l'intercepteur finalisé, l'idée est de développer un module qui serait intégré à SimGrid et chargé de traiter les informations renvoyées par l'intercepteur, comme le montre le schéma suivant :

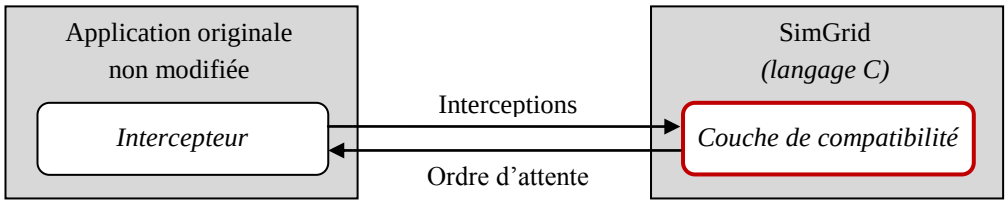


Fig.6 : Fonctionnement global de l'intercepteur avec SimGrid

Lorsque l'intercepteur recevra une demande de communication de la part de l'application, il en fait part au module de SimGrid chargé de la traiter et qui joue donc le rôle de proxy. Selon le type de communication, SimGrid demandera alors (via le proxy) à l'application d'attendre un certain temps ou d'avancer son horloge virtuelle, simulant le temps qu'auraient nécessité les calculs à faire par exemple.

Or, comme SimGrid est essentiellement codé en C et que l'intercepteur l'est en Java, il sera nécessaire d'utiliser les *bindings* de l'API MSG pour assurer la connexion inter-langage.

Malheureusement, comme nous n'avons pas eu le temps d'implémenter dans l'intercepteur un moyen propre de renseigner l'extérieur à chaque fois que l'application effectuerait une action (envoyer ou recevoir une communication, exécuter un calcul), le développement du proxy n'a pas été réalisé.

Pour nous habituer au fonctionnement du simulateur, nous avons simplement commencé par tester les *bindings* directement dans une de nos applications (All-to-All) et avons alors travaillé à partir d'instructions lues dans un fichier source qui remplissait le rôle qu'aurait eu l'intercepteur s'il avait été finalisé.

III. Difficultés rencontrées

Difficultés au cours des réalisations

Les difficultés que nous avons rencontrées sont essentiellement techniques car nous ne possédions à l'origine pas toutes les connaissances utilisées au cours du projet. Par exemple, nous avions l'habitude de travailler dans l'environnement d'Eclipse et quand il a fallu rendre nos applications exécutables depuis la console avec des lignes de commandes, nous avons du nous familiariser avec les problèmes de *classpath* qui étaient auparavant gérés automatiquement. De même, et comme il a été dit précédemment, la programmation d'applications distribuées était nouvelle pour nous. Nous avons ainsi connu des difficultés pour finaliser nos applications proprement, notamment lors de la rupture de communication entre le serveur et ses clients.

Difficultés sur la gestion de projet

Les écarts de prévisions que nous avons eus entre l'échéancier initial et le travail effectivement réalisé sont assez flagrants et matérialisés sur le diagramme de Gantt de la page suivante. On y remarque que l'échéancier initial prévoyait un travail séquentiel alors que nous avons, de fait, plutôt procédé à une démarche en parallèle pour faire face aux retards et essayer de tenir au mieux le cahier des charges. Plusieurs éléments peuvent expliquer ces différences :

D'une part, nous avons commencé à prendre du retard dès la première phase d'implémentation des applications distribuées, imputable aux difficultés citées ci-dessus. Bien que les applications aient été prêtes à la date indiquée de l'échéancier, nous n'avions pas prévu de les rendre accessibles en dehors d'Eclipse (évolution ver un lancement à partir de lignes de commandes)…

D'autre part, notre projet s'inscrivait dès le départ dans un axe de recherche. Or, la recherche est un travail qui nécessite beaucoup de temps et surtout, une quantité fluctuante de temps. Lors de l'établissement même de l'échéancier au début d'année, il était donc déjà difficile de prévoir le travail qui aurait été nécessaire à l'exploration de chacune des pistes pour résoudre un problème donné. En effet, le principal outil que nous avons utilisé, les agents java, est de manière générale assez peu documenté, notamment pour l'usage que nous voulions en faire.

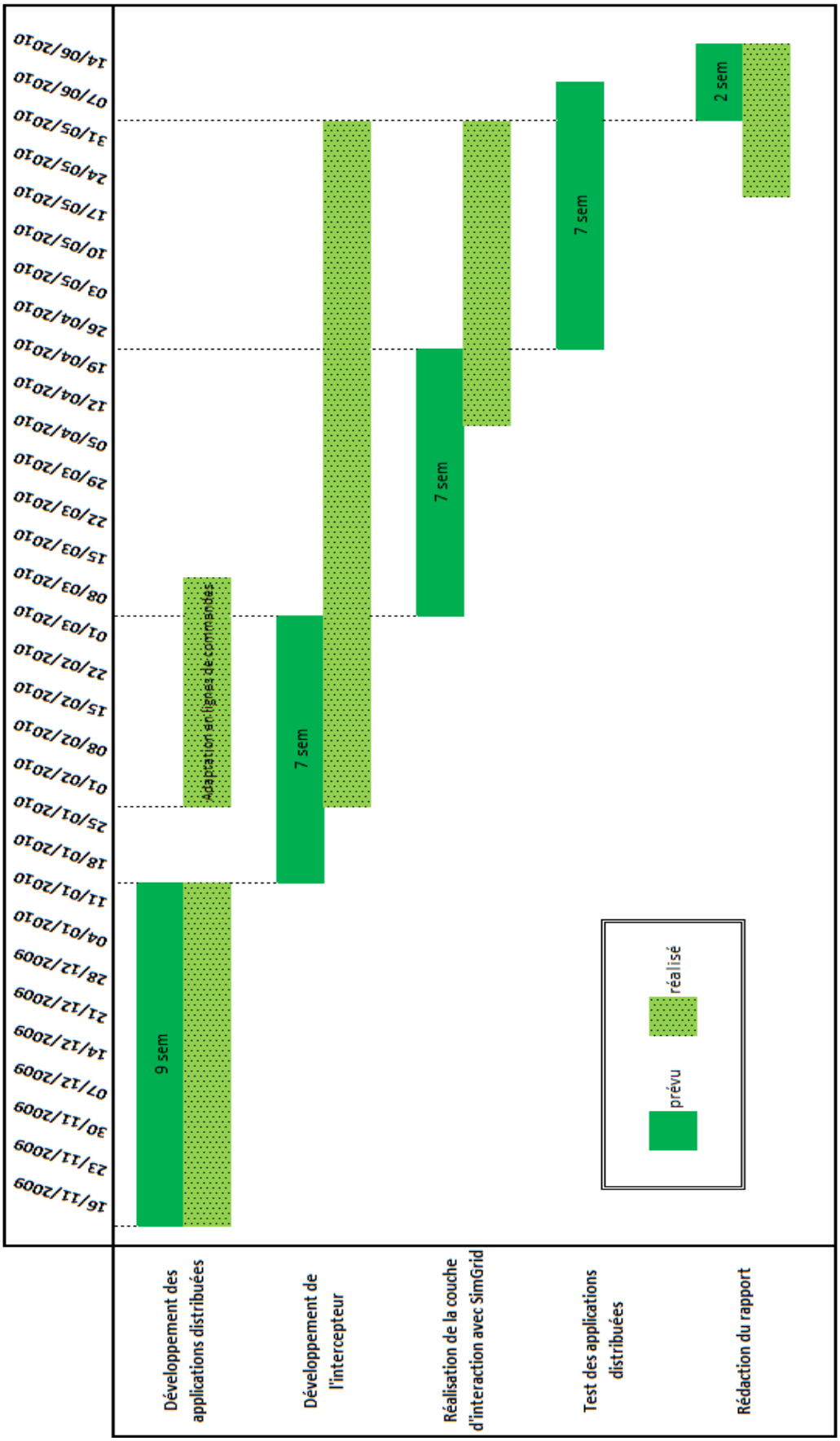


Fig.7 : Diagramme de Gantt

Conclusion

Tout d'abord, nous souhaitons remercier nos tuteurs de projet pour l'aide qu'ils nous ont apporté tout au long de l'année, pour avoir répondu à nos questions et pour avoir eu la gentillesse de nous conseiller dans la rédaction du présent rapport.

Notre bilan du projet par rapport à ses objectifs est en demi-teinte. Nous n'avons certes pas rempli l'ensemble des fonctions attendues au cahier des charges : la finalisation de l'intercepteur et l'implémentation d'un proxy pour SimGrid sont toujours d'actualité. Néanmoins, nous avons indéniablement beaucoup appris, depuis les applications distribuées jusqu'à l'aperçu que nous avons pu avoir de ce à quoi ressemble un sujet de recherche. Il a été l'occasion de découvrir de nouvelles notions informatiques, complétant notre formation des Mines et, dans la continuité du projet de 1ère année, nous avons de nouveau constaté qu'il était parfois difficile de tenir des délais car des difficultés imprévues sont toujours possibles.

Nous pensons par ailleurs que notre travail au cours de cette année servira de bases solides pour les étudiants éventuels qui reprendront le flambeau. A cette fin d'universalité et pour assurer la meilleure passation du projet possible, nous avons commenté les codes sources du mieux possible et en anglais.

Annexes

Il peut être plus simple pour la compréhension des annexes de disposer du code source des applications développées sous les yeux. A cette fin, ces derniers sont accessibles ici :

http://subversion.assembla.com/svn/Algorille

A. Les différentes applications distribuées développées

A chaque fois, nous invitons le lecteur à se référer au code commenté pour avoir des précisions sur l'implémentation.

La classe Message

Les applications distribuées qui ont été développées utilisent toutes le protocole TCP via la classe *Socket*.

Dans le cadre de notre projet, il ne s'agissait pas d'échanger de simples chaines de caractères. Pour remplir les objectifs que nous nous étions fixés, il était nécessaire de pouvoir identifier la nature des communications échangées ainsi que leur taille. Nous avons donc créé pour chaque application une classe *Message* qui est chargée de rassembler ces informations.

Cette classe permette d'envoyer des objets d'une classe à l'autre via l'utilisation des classes *ObjectOutputStream* et *ObjectInputStream*. Les *Message* envoyés doivent néanmoins être persistants et c'est à cette fin qu'ils implémentent l'interface *java.io.Serializable* qui permet de s'assurer qu'une autre JVM restituera correctement l'état de l'objet.

Intérêt du multithreading

L'intérêt de la simulation est de pouvoir concentrer l'exécution de plusieurs machines virtuelles sur un même ordinateur. Il faut donc qu'une application donnée, par exemple le *Scatter-Gather*, puisse être lancée plusieurs fois sur une même machine physique. Il suffit donc de donner en argument au programme une adresse IP qui sera transmise ensuite en argument au constructeur de la classe *Socket*.

Néanmoins, chaque application doit à la fois être capable d'envoyer des messages et de les recevoir, et ce vis-à-vis d'éventuellement plusieurs nœuds. La nécessité du multithreading intervient ici avec un *Thread* réalisant chacune de ces fonctions. Pour réaliser le multithreading, il était possible soit d'implémenter l'interface *Runnable* qui impose une méthode *run()*, puis d'instancier un nouveau *Thread* et de le lancer, soit de faire hériter une machine d'un *Thread*.

Ping-pong : le modèle du Client/Serveur

C'est le modèle le plus simple de réseau qui puisse exister : un serveur et un unique client. Le serveur doit pouvoir fonctionner en permanence de telle sorte que si un client cherche à se connecter, il trouve un hôte auquel adresser sa demander.

On ne fonctionne donc ici qu'avec deux machines : un serveur et un client. A la connexion du client sur le serveur, les messages sont automatiquement envoyés au serveur pour que nous puissions mesurer le temps de transmission à travers le réseau.

Token Ring : structure en anneau

Ce modèle place les machines en anneaux. Chaque ordinateur a donc deux voisins, que l'on peut décrire comme un voisin de gauche et un voisin de droite. Lors de l'implémentation, nous avons défini dans un premier temps que l'information ne proviendrait que du voisin de gauche et qu'elle ne sera transmise qu'au voisin de droite.

La principale difficulté réside dans la fermeture du réseau. Elle a été résolue en donnant en argument au lancement du *MainTokenRing* la position de l'ordinateur se connectant : premier, milieu ou dernier. Ce n'est donc pas un réseau « auto-bouclant » au sens où il faut déjà connaître sa taille avant de clore le circuit. Selon la position du nœud créé, le nombre d'arguments peut varier et le programme appelle le constructeur adéquat en fonction de ce nombre.

All-to-All

Dans une structure All-to-All, tout le monde communique avec tout le monde. De ce fait, chacun est au courant de tout ce qui se passe.

La principale difficulté a été de pouvoir gérer la connexion supplémentaire d'un nœud après que le réseau ait été établi. Nous avons levé ce problème en créant une classe *Network* qui est transmise à chaque nouvelle connexion et qui contient toutes les informations relatives au réseau dans des *HashMap*.

De même que dans le Token Ring, selon le nombre d'arguments au lancement du programme, différents constructeurs peuvent être appelés pour soit créer un nouveau réseau, soit rejoindre un réseau existant.

Scatter-Gather

Une requête est envoyée par un client à un serveur qui redistribue (*scatter*) les données à l'ensemble de clients. Ceux-ci rendent compte au serveur une fois les calculs effectués. Le serveur rassemble (*gather*) alors les données pour fournir le

résultat global au demandeur. L'intérêt est bien sûr d'utiliser les ressources de plusieurs machines pour calculer plus rapidement.

Alors que le serveur est unique, deux cas se présentent pour le client : soit il demande à effectuer une opération, soit il partage ses ressources. La distinction s'effectue au lancement du client par le nombre d'arguments qui lui sont passés.

B. Compléments techniques sur l'intercepteur

Nous développons ici quelques points qui méritent explication mais qui ne peuvent l'être dans les commentaires. Des éléments complémentaires succincts sont aussi repris ici. Globalement, l'intercepteur tourne autour des quatre points ci-après :

Le Manifest et le fichier jar

Un agent java est un argument. En fait, on lance un *main* via la console. De ce fait, nous appelons la commande java qui fait intervenir un argument ***javaagent***. Les lignes de commandes sont dans les fichiers bat respectifs.

Nous avons donc besoin d'un fichier jar qui contient tout l'intercepteur. Conformément à la portabilité requise dans les objectifs, on souhaite avoir le moins possible à exporter. Un fichier jar parfait qui contient tout serait l'idéal.

Pour interpréter un fichier jar, java lit le ***manifest*** qui est une sorte de guide au sein du fichier compressé. Il faut donner la moindre précision : à partir de quel fichier on exécute l'intercepteur ? Quelles sont les autorisations de l'agent ?

Le fichier Agent.java

Notre classe agent implémente la classe ***ClassFileTransformer*** dont la méthode principale est la méthode ***transform***. Cette méthode est très puissante car en plus de pouvoir filtrer un *bytecode* (une classe est chargée, interceptée puis transformée), elle peut réitérer ce processus en fonction des autres événements. Une classe peut être amenée à être modifiée plusieurs fois ce qui n'est pas notre cas.

Les agents exigent une extrême rigueur. En effet, des détails simples (les noms de classes et packages ne s'écrit pas séparés par des points mais par des back-slash « / »). Les noms de classes sont vérifiés à chaque chargement de classes et deux classes ne peuvent avoir le même nom, c'est pourquoi il y a un *Test.class*, *MyTest.class* et *RealTest.class*. Cependant, le nom du fichier (qu'on voit dans l'explorateur) n'est pas forcément le même que le nom de classe. *MyTest.class* et *Test.class* sont tous les deux des fichiers de la classe Test.

ASM : ClassRenamer.java

Prenons le cas de la classe *Test*. On intercepte la classe *Test* et on la remplace par *MyTest.class*. On stocke la vraie classe *Test*. Mais *MyTest.class* fait appel à *RealTest.class* qui après interception renvoie le vrai code source de la classe. Comme deux classes différentes ne peuvent avoir le même nom, on est contraint de renommer la vraie classe *Test*, qui est alors un attribut de la fausse classe *Test*, en *RealTest*.

Or, renommer une classe n'est pas aisé. Il suffit de voir ce qui se passe dans Eclipse lors qu'on appelle la commande *Rename* (dans *Refractor*). Il renomme la classe (public class *RealTest*) mais aussi toute les occurrences de *Test* en *RealTest*, et ce avant compilation.

Notre cas concerne une classe déjà compilé. Il faut donc renommer toutes les occurrences de classes (dans les attributs, dans les méthodes) de *Test* en *RealTest*. Nous n'avons pas trouvé d'autres solutions que d'utiliser ASM qui est un puissant décompilateur.

L'intérêt d'ASM réside dans la classe ***ClassRenamer*** qui comme son nom l'indique renomme les classes. Basiquement, on a **un *ClassReader*** qui « charge » une classe. Un ***ClassVisitor*** visite, c'est-à-dire parcourt, le *bytecode* de la classe chargé. A travers la *ClassVisitor*, on peut ajouter un *ClassAdaptor* qui est une implémentation de *ClassVisitor*. ***ClassAdaptor*** peut modifier, ajouter, supprimer du *bytecode*. Mais l'opération peut être très laborieuse. Après toutes ces modifications avec les bons adaptateurs, une ***ClassWriter*** se charge de reconvertir le résultat en un *bytecode* propre et lisible par la JVM.

Message.java et RealMessage.java

Nous avons vu précédemment que *RealMessage.java* est un attribut de la fausse classe *Message*. On, parle de **délégation objet**. En effet, *Message.java* se charge du travail d'intercepteur, de modifier les informations si nécessaire alors que *RealMessage.java* exécute aveuglément les vraies tâches de la vraie classe *Message*.

Il nous suffit alors d'écrire les méthodes concernés et les constructeurs concernés. Pour cela, il est nécessaire de savoir quelles sont les méthodes appelées. Par exemple, la méthode *accept()* de la classe *ServerSocket* fait appel au constructeur *Socket* (*SockImpl impl*). Pour cela il suffit de voir dans le fichier source de la classe *ServerSocket*. **Télécharger la documentation** ou avoir un décompilateur est alors indispensable.

D'autre part, un élément important est le ***débugger* étape par étape d'Eclipse**. Exécuter un programme étape par étape permet de voir quelles sont les classes et méthodes appelées. On peut y modifier les points d'arrêt, choisir de revenir en arrière, de sauter une étape…