MajecSTIC 2009
Avignon, France, du 16 au 18 novembre 2009

Dagda, un intergiciel pour la distribution dynamique de simulation de système complexe.

Guilhelm Savin[1]

1 : Université du Havre, LITIS, rue Philippe Lebon, 76600 Le Havre - France.

Contact : guilhelm.savin@litislab.fr

Résumé
Les systèmes complexes qui sont modélisés et simulés en informatique deviennent de plus en plus sophistiqués. La puissance de calcul d'une simple machine devient insuffisante pour exécuter ces simulations. Une solution consiste à distribuer la simulation afin d'utiliser la puissance de calcul d'un ensemble de machines.
DAGDA, l'architecture et la plateforme présentées dans ce papier, offre une couche entre la simulation d'un système complexe et les ressources de calcul disponibles. Cette couche gère la répartition des entités sur les machines de façon à équilibrer les charges de calcul de chaque machine et réduire les communications entre les machines.

Abstract
Complex systems which are modeled and are simulated in computer science become increasingly sophisticated. The computing power of a single machine becomes insufficient to execute these simulations. Therefore, it needs to exploit the computing power of a set of machines.
DAGDA, the architecture and the platform which are presented in this paper, offers a layer between simulation of a complex system and the available resources. This layer manages spreading of entities on machines to reduce work-load and network-load of each machine.

Mots-clés : intergiciel, équilibrage de charges dynamique, simulations de systèmes complexes

Keywords: middleware, dynamic load-balancing, complex systems simulations

1. Introduction

Les programmes demandant de plus en plus de ressources de calculs, les développeurs se tournent vers la programmation distribuée qui permet d'exploiter la puissance de plusieurs machines. Ce type de programme soulève cependant certains problèmes. La communication entre les différentes composantes du programmes est un problème majeur : comment réaliser une couche permettant l'appel de méthodes distantes tout en réduisant l'impact sur les performances ? Les architectures et les sytèmes d'exploitation des différentes machines peuvent aussi soulever un problème : est-il possible d'avoir des architectures ou des systèmes d'exploitation différents ? Un autre problème tout aussi important concerne le choix de la politique de distribution des différentes composantes du programme distribué.
Dans cet article nous nous concentrons sur les simulations de systèmes complexes et nous proposons une plateforme, DAGDA, dédiée à leur distribution. Ce type de simulations est souvent constitué d'un ensemble massif d'entités avec de nombreuses interactions entre ces dernières. L'entité est un concept générique permettant d'englober différentes représentations telles que celles d'agent et d'objet. L'exécution de telles simulations peut être modélisée par un graphe qui évolue dans le temps et qui permet de réprésenter les interactions (les arêtes du graphes) existantes dans un ensemble d'éléments (les nœuds).
DAGDA fusionne un intergiciel, qui permet la communication et la migration des entités, et un répartiteur de charges permettant d'établir une politique de répartition des entités sur les machines.

Les intergiciels forment une catégorie de programmes qui créent une couche entre une application distribuée et des ressources de calcul. Ils aident les développeurs en créant une abstraction des ressources, ce qui les décharge de la gestion des ressources et des problèmes associés et leur permet ainsi de se concentrer sur l'application. Les intergiciels sont décrit dans la sous-section 1.2. DAGDA utilise l'algorithme de répartition de charge $AntCo^2$ [4] qui est décrit dans la partie 2. Il a été choisi car il répartit les entités en considérant non seulement la charge des machines mais aussi les interactions existantes entre les entités. Le concept de répartition de charge est décrit dans la partie 1.3.

1.1. Objet actif

Un concept important qu'il est nécessaire de présenter pour la suite de ce papier est le concept d'*objet actif* [9]. La différence entre un objet et un objet actif se situe entre l'appel d'une méthode de l'objet et son exécution. Avec les objets de base, l'appel et l'exécution de la méthode sont synchrones et se situent dans le même fil d'exécution comme le montre la figure 1.

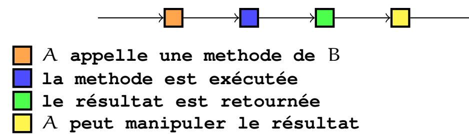

FIGURE 1 – appel d'une méthode d'un objet

Avec les objets actifs, l'appel et l'exécution sont asynchrones. Les appels aux méthodes sont des requêtes envoyées à l'objet actif qui possède son propre fil d'exécution afin de les traiter (cf. figure 2). Il enregistre les requêtes dans une liste et les traite selon une politique définie.

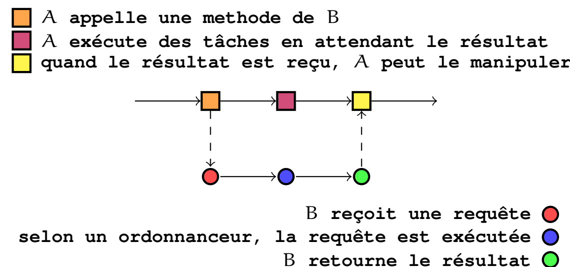

FIGURE 2 – appel d'une méthode d'un objet actif

La figure 3 décrit ce processus : l'émetteur, celui qui appelle la méthode, envoie une requête au travers d'un objet *talon* [1] à un objet actif. La communication entre le talon et l'objet actif s'effectue au travers d'un proxy. Un *futur* est retourné immédiatement après l'envoi de la requête mais ne sera utilisable qu'une fois la requête exécutée. Ce *futur* est une répresentation du retour de la méthode. L'intêret des objets actifs est que l'émetteur peut exécuter d'autres tâches en attendant l'activation du *futur* : l'appel aux méthodes n'est pas bloquant.

1.2. Intergiciel

Un intergiciel fournit une connexion entre des logiciels ou entre les composants d'un logiciel. Cette connexion permet la communication entre des processus. Ces derniers peuvent être localisés sur la même machine ou sur différentes machines connectées sur un même réseau. Par

1. *stub object*

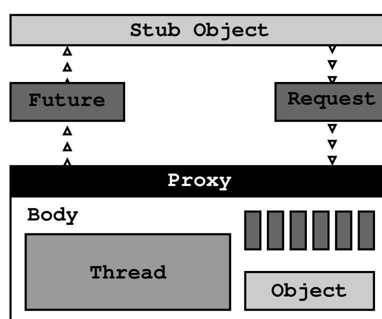

FIGURE 3 – Objet actif

conséquent, l'intergiciel permet d'exploiter la puissance de plusieurs machines. Il permet aussi de gérer la connexion et déconnexion des machines qui participent à l'exécution du programme. Il crée donc une grille dynamique de machines pouvant être utilisée pour exécuter une application distribuée.
Il existe différents types d'intergiciels. Certains fournissent un moyen de soumettre des tâches à d'autres machines qui retourneront le résultat par la suite. Dans ce cas, il n'y a pas d'interactions entre les différentes machines participantes et c'est habituellement une approche centralisée : une machine maitresse envoie des tâches à d'autres machines esclaves. BOINC [1] est un exemple de ce type d'intergiciel, il est utilisé dans les projets `@home` tel que `seti@home` [2].
D'autres ont une approche objet de la distribution et certains d'entre eux utilisent le concept d'objet actif. C'est le cas de ProActive [2] développé par l'INRIA à Sophia Antipolis. Ce type d'intergiciel permet les interactions entre les objets distribués.
Parmis les fonctionnalités que fournissent les intergiciels, l'espace d'adressage unique est l'une des plus importantes. En effet, avec un programme non-distribué, l'affectation d'un identifiant unique à chaque objet n'est pas un problème : l'adresse mémoire en est un. Cependant, lorsque les objets sont répartis sur un ensemble de machines, obtenir un identifiant unique devient un problème, les adresses mémoires ne pouvant plus satisfaire ce rôle.

1.3. Répartition de charge

Le concept de répartition de charge peut être décrit de la façon suivante : en considérant $\mathbb{S}$ l'ensemble de machines et $\mathbb{T}$ l'ensemble de toutes les tâches exécutables, alors la répartition de charge est une fonction $l : \mathbb{T} \to \mathbb{S}$ qui à une tâche t attribue une machine $l(t) = m$, $(m \in \mathbb{S})$. Il permet donc de définir une politique de répartition des tâches sur l'ensemble des machines.
On peut citer par exemple :
– la répartition de charge équitable (Round-robin) et sa variante répartition de charge pondérée (Weighted Round-robin [6]) ;
– la répartition de charge selon la moindre connexion (Least-connection) ;
– la répartition basée sur la charge (Load-based).
Ce concept est utilisé par les services web par exemple, pour répartir les requêtes des utilisateurs entre plusieurs serveurs : l'utilisateur voit l'ensemble des serveurs comme un seul, quand il envoie une requête, le répartiteur de charges choisit vers quel serveur rediriger cette requête. La charge des serveurs est ainsi équilibrée, ce qui permet d'offrir une meilleure qualité de service à l'utilisateur.
En programmation distribuée, il permet d'optimiser la charge de calcul des machines en établissant une politique de répartition des tâches. Le répartiteur de charges peut dépendre du type de réseau qui peut être synchrone ou asynchrone, et dont la topologie peut être dynamique.

2. `http://setiathome.ssl.berkeley.edu/`

1.4. Graphes dynamiques

L'exécution des simulations distribuées par DAGDA peut être réprésentée par un graphe dynamique, nous allons donc introduire le concept de graphe puis celui de graphe dynamique.

Un graphe G est une paire (N, E) où N est un ensemble d'éléments appelés *nœuds* et E est un ensemble de paires (u, v) de nœuds appelées *arêtes* tel que $u, v \in N$. Un graphe permet donc de décrire des liens entre des éléments. Un graphe dynamique est une suite $G_i = (N_i, E_i)$ de graphes telle que $\forall(u, v) \in E_i, u, v \in N_i$. C'est donc un graphe qui peut varier dans le temps par l'ajout ou la suppression de nœuds ou d'arêtes. Les graphes dynamiques permettent de décrire l'évolution des interactions dans le temps entre des éléments.

Les nœuds représentent les entités des simulations et les arêtes représentent les interactions entre ces entités. Les arêtes du graphe modélisant l'exécution de la simulation sont pondérées afin de quantifier l'importance de l'interaction : le poids est proportionnel à la fréquence de l'interaction.

2. AntCo^2

AntCo^2 est un algorithme distribué dédié à la répartition de charge et à la minimisation des communications. Il considère le graphe dynamique représentant l'exécution de l'application dont on souhaite obtenir une distribution.

Comme les interactions entre les entités apparaissent et disparaissent, et que l'importance de l'interaction évolue, le graphe change. Par conséquent, le répartiteur de charge doit aussi manipuler ce processus dynamique et être capable de fournir une distribution tant que le graphe évolue.

Chaque ressource de calcul est associée à une couleur, ensuite en assignant une couleur à un nœud (une entité) l'algorithme spécifie la distribution.

Certains voient la distribution comme un partitionnement pondéré du graphe [10]. Dans ce partitionnement nous essayons de distribuer uniformément la charge (nombre d'entités pondérées par leur demande de ressources) et de minimiser les communications entre les machines afin d'éviter la saturation du réseau. Comme ces deux critères sont conflictuels, il est nécessaire de trouver un compromis.

Nous voyons le partitionnement comme un algorithme de détection dynamique de communautés. Ces communautés dynamiques sont appelées *organisations*. Les communautés sont souvent vues comme un groupe de nœuds dont les connexions entre les membres sont plus denses qu'avec le reste du graphe. Un algorithme capable de détecter les organisations est capable de suivre l'évolution des communautés quand les nœuds ou les arêtes apparaissent, évoluent et disparaissent.

Il existe plusieurs algorithmes de partionnement de graphe ([7,8,10]) et de détection de communautés ([11]), mais peu sont capables de gérer l'évolution du graphe. Il est toujours possible de redémarrer l'algorithme à chaque changement du graphe, mais ceci implique un calcul intensif. AntCo^2 est un algorithme incrémental qui utilise le partionnement précédent pour en calculer un nouveau lorsque le graphe change.

Avoir un répartiteur de charge tournant sur une seule machine, pour distribuer des applications qui sont souvent très importantes peut se révéler inefficace. Un autre but de AntCo^2 est d'être capable d'être distribué avec l'application.

AntCo^2 utilise une approche fondée sur l'intelligence en essaim s'appuyant sur la métaphore naturelle des colonies de fourmis. Cet algorithme apporte plusieurs avantages : les fourmis peuvent agir avec seulement une connaissance locale du graphe représentant l'exécution de l'application à distribuer, ce qui lui permet d'être lui-même distribué avec peu de communications et sans contrôle global.

Dans AntCo^2, chaque colonie représente une ressource de calcul et possède sa propre couleur. À l'intérieur des colonies, les fourmis collaborent pour coloniser les organisations à l'intérieur du graphe et assignent leur couleur aux nœuds. Inversement, les colonies sont en compétition pour garder et conquérir des organisations.

Les fourmis colorent les nœuds en utilisant des phéromones numériques dont la couleur correspond à celle de leur colonie. Ces phéromones *s'évaporent* et par conséquent doivent être constamment maintenues par les fourmis. Ce phénomène permet de gérer la dynamique du graphe en oubliant les anciennes solutions de partionnement pour en découvrir de nouvelles par une exploration constante du graphe par les fourmis. Les détails de l'algorithme sont donnés dans [3].

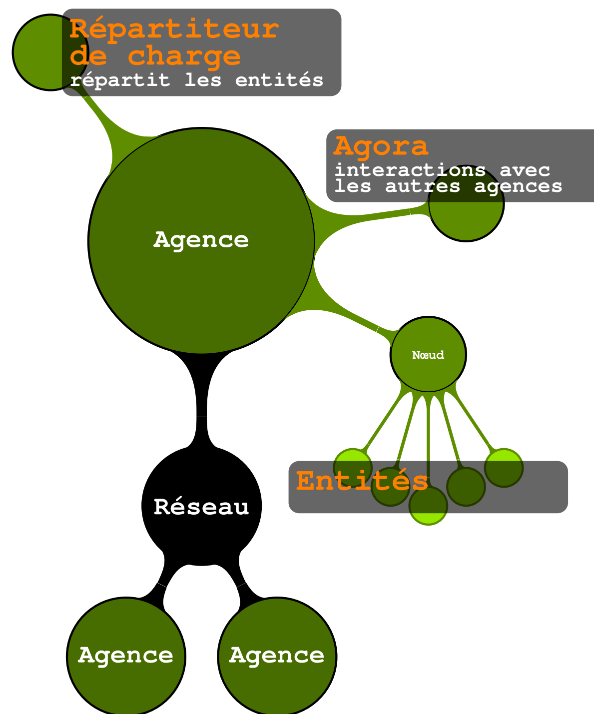

FIGURE 4 – Vue d'ensemble de DAGDA

Le changement de la couleur d'un nœud indique un *conseil de migration*, signifiant que l'entité correspondante devrait migrer vers la ressource de calcul associée à la couleur. Un mécanisme d'inertie permet d'éviter les conseils oscillatoires.

3. Dagda

DAGDA est un intergiciel dédié à la distribution de simulations de système complexe. Il utilise un intergiciel existant auquel il ajoute de nouvelles fonctionnalités. Le but final est de fournir un moyen simple de créer des simulations de systèmes complexes.
Les maîtres mots caractérisant DAGDA sont *décentralisé, portable* et *réparti*. Décentralisé signifie qu'il n'y a pas un ensemble restreint de machines dont dépend toutes les autres. DAGDA se veut le plus portable possible, c'est à dire que tout type de machines (ordinateur,pda,téléphone,super-calculateur,...) peut participer à la distribution.
L'intergiciel actuellement utilisé est ProActive [2]. Ce choix est motivé par l'approche *objets actifs* utilisée dans ProActive.

3.1. Entités

DAGDA est basé sur le concept que l'application à distribuer est composée d'un ensemble massif d'entités. Ces entités sont des objets actifs et sont hébergées par une machine que l'on nommera *agence*.
Les entités peuvent interagir entre elles et peuvent migrer d'une agence à l'autre. Cela soulève certains problèmes : comment identifier chaque entité à travers le réseau et comment contacter une entité distante ? Le contact d'une entité est traité dans la partie 3.2. Chaque entité possède un identifiant qui dépend de la date de création et de l'adresse de l'agence créant l'entité ce qui le rend unique sur le réseau et dans le temps.

3.2. Communication entre les agences

DAGDA se veut décentralisé, par conséquent il n'y a pas de serveur maître utilisé comme annuaire pour référencer les entités et leur localisation. Il est donc nécessaire d'introduire certains

mécanismes permettant de fournir des fonctionnalités comme la recherche d'une entité.
Ce rôle est assumé par un composant de DAGDA qui permet de détecter et de se faire détecter par d'autres agences, puis d'échanger des informations afin de trouver une entité par exemple.

3.3. Contexte

Un programme possède certains paramètres qui créent un contexte utilisable par les composants de ce programme. Lorsque le programme est constitué d'un seul processus, il y a partage de la mémoire et donc du contexte. Cependant lorsque le programme est distribué, chaque machine possède sa propre mémoire et le partage de paramètres globaux devient un problème.
DAGDA crée un contexte divisé en deux parties. La première partie est *locale* et contient les paramètres propres à la ressource de calcul. La seconde est *globale* et les changements sont diffusés à l'ensemble des ressources.

3.4. Graphe d'interaction

DAGDA étudie les appels de méthodes entre entités. Par exemple, si une entité `A` appelle une méthode `m()` d'une entité `B`, cet appel est détecté et enregistré. Ensuite, cette détection d'interactions entre entités est utilisée comme fournisseur d'événements d'un graphe dynamique qui modélise ces interactions à travers le temps. Les nœuds du graphe sont les entités hébergées par la machine ou les entités distantes étant en interaction avec une entité locale. Les arêtes du graphe représentent les interactions et le poids de ces arêtes est associé à la fréquence de l'interaction. Un mécanisme fait décroître le poids des arêtes dans le temps ce qui permet de maintenir une cohérence dans la signification des arêtes.
La gestion du graphe se fait en utilisant l'API GRAPHSTREAM [5] [3]. Le graphe peut ensuite être utilisé par les composants de DAGDA comme le répartiteur de charges.

3.5. Répartiteur de charges

Les entités sont réparties sur les machines disponibles à l'aide de l'algorithme de répartition de charges AntCo^2. Ce choix permet :
- d'équilibrer la charge des machines ;
- réduire la charge du réseau ;
- distribuer le répartiteur de charges.

La distribution du répartiteur de charges est un point important pour obtenir une plate-forme décentralisée. Dans [4], trois méthodes d'exécution de AntCo^2 sont présentées. La première et la seconde consistent à exécuter l'algorithme sur un ensemble restreint de machines dédiées (une seule machine pour le premier cas). Dans ces deux cas, l'ensemble de la charge de calcul des serveurs est dédié à l'exécution de AntCo^2 et l'algorithme a une vue globale de l'application distribuée. Le dernier cas utilise chaque machine disponible pour exécuter l'algorithme. Dans ce cas, AntCo^2 utilise peu de la capacité de calcul des machines et chaque instance de l'algorithme a une vue locale de l'application distribuée. Ceci permet de décentraliser AntCo^2 et c'est cette voie qui a été choisie pour DAGDA.
Comme la charge de calcul dédiée à AntCo^2 est fonction du nombre d'entités présentes sur la machine, distribuer les entités sur l'ensemble des machines revient à distribuer AntCo^2 lui même : AntCo^2 est auto-distribué.

4. Résultats

La plateforme DAGDA est capable de créer les entités et d'analyser les interactions entre elles afin de générer un *graphe d'interactions* représentant l'exécution de la simulation en temps réel. Elle est aussi capable de connecter les agences entre elles et de migrer les entités d'une agence à l'autre.

4.1. Application de test

Afin de pouvoir réaliser des tests sur DAGDA, une application simple a été écrite dont le but est de générer des entités, des interactions entre entités et des migrations entre agences. Les entités utlisées pour cette application sont décrites de la façon suivante :

3. `http://www.graphstream-project.org`

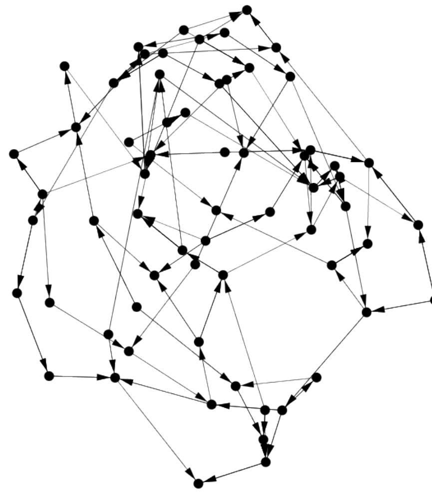

FIGURE 5 – Exécution du programme de test

```
TESTENTITY :
  attributes :
    List<TestEntity> neigh
  methods :
    call( TestEntity te ) {
      while( neight.size() > MAX )
        neigh.poll();
      neigh.add(te);
    }
    execute() {
      int i,j;
      i = random() % neigh.size();
      j = random() % neigh.size();
      neigh.get(i).call(neigh.get(j));
      if( random() < P_MIGRATION )
        migrateSomewhere();
    }
```

L'application crée un ensemble de `TestEntity` et initialise aléatoirement l'attribut *neigh* représentant une liste d'entités avec lesquelle il y aura des interactions. Ensuite, chaque agence exécute en boucle la méthode *execute()* de chaque entité qu'elle héberge.
La figure 5 représente le graphe de l'exécution de cette application avec un ensemble de 64 entités.

5. Conclusion

Dans cet article, nous avons présenté les concepts d'intergiciel et de répartiteur de charges. Ensuite nous avons présenté la plateforme DAGDA qui fusionne un intergiciel et l'algorithme de répartition de charge AntCo^2.
DAGDA est toujours en développement mais est capable de lancer des simulations et d'analyser leur exécution. La prochaine étape consiste à finaliser l'implémentation du répartiteur de charges et de valider la plateforme en effectuant une batterie de tests.
Ensuite, il sera nécessaire de fournir une simulation de démonstration pour DAGDA et de réaliser des tests de performances afin de présenter le gain apporté par la plateforme.

Bibliographie

1. David P. Anderson. Public computing : Reconnecting people to science. In *Conference on Shared Knowledge and the Web*, Residencia de Estudiantes, Madrid, Spain, Nov. 2003.
2. Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, et Romain Quilici. *Grid Computing : Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.
3. Cyrille Bertelle, Antoine Dutot, Frédéric Guinand, et Damien Olivier. Organization detection for dynamic load balancing in individual-based simulations. *Multi-Agent and Grid Systems*, 3(1) :42, 2007.
4. Antoine Dutot. *Distribution Dynamique Adaptative à l'aide de mécanismes d'intelligence collective*. Thèse de doctorat, Université du Havre - LIH, 2005.
5. Antoine Dutot, Frédéric Guinand, Damien Olivier, et Yoann Pigné. Graphstream : A tool for bridging the gap between complex systems and dynamic graphs. In *EPNACS : Emergent Properties in Natural and Artificial Complex Systems*, 2007.
6. M. Katevenis et al. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE J. of Selected Areas in Comm.*, 9(8) :1265–1279, 1991.
7. C. M. Fiduccia et R. M. Mattheyses. A linear time heuristic for improving network partitions. In *ACM IEEE Design Automation Conference*, pages 175–181, 1982.
8. B. Hendrickson et R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Scien. Comput.*, 16(2) :452–469, 1995.
9. Carl Hewitt, Peter Bishop, et Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
10. B.W. Kernighan et S. Lin. An efficient heuristic procedure for partitioning graph. *The Bell System Technical Journal*, 49(2) :192–307, 1970.
11. M. E. J. Newman et M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev*, 69, 2004.