HAL
open science

Dagda, a load-balanced middleware to distribute Complex Systems simulations

Antoine Dutot, Damien Olivier, Guilhelm Savin

To cite this version:

Antoine Dutot, Damien Olivier, Guilhelm Savin. Dagda, a load-balanced middleware to distribute Complex Systems simulations. 3rd International Conference on Complex Systems and Applications (ICCSA'09), Jun 2009, Le Havre, France. pp.171-174. hal-00448387

HAL Id: hal-00448387

https://hal.science/hal-00448387v1

Submitted on 18 Jan 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Dagda, a load-balanced middleware to distribute Complex Systems simulations

Antoine Dutot, Damien Olivier, Guilhelm Savin

***Résumé*—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 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.

***Index Terms*—middleware, dynamic load-balancing, complex systems simulations**

I. INTRODUCTION

PROGRAMS asking increasingly computing ressources, developers create distributed programs that aim to be executed on several computers. This kind of programs raises some problems. There are problems about communication between remote parts of the program : how to realize a layer to process remote calls, what impact this layer will have on the program's performances. There are also problems about computers architecture : is same architecture needed for computers, etc... Finally, there is problem of how distribute tasks or components of the distributed program.

In this paper, we focus on simulations of complex systems and we propose a dedicated platform, DAGDA, for their distribution. This kind of simulations is often composed of a massive set of entities with many interactions between them. Entity is a generic concept which covers for example agent and object concept. Execution of an entity is not deterministic because of interactions existing between the entity and other entities or environment, what explains a distributed approach rather than others methods used in deterministic programs. Execution of such simulations can be modeled by a dynamic graph which allows to describe interactions (edges) existing in a set of elements (nodes).

DAGDA merges a middleware, to allow communication between remote entities, and a load-balancer, to spread these entites on available machines.

Middlewares are a category of programs that creates a layer between a distributed application and computing ressources. They help developers by creating an abstraction of the ressources, so developers do not have to deal with resource problems and can focus on the application. The subsection I-B describes middlewares.

DAGDA uses the load-balancing algorithm ANTCO^2 which is described in II. It has been chosen because it spreads entities considering not only the workload of machines but also interactions existing between entities. The load-balancing concept is described in subsection I-C.

A. Active Object

An important pattern needing to be presented for this paper is the active object pattern[1]. The difference between a *basic* object and an *active* object takes place between method call and its execution. With basic object, calling and execution of a method are synchronous (ie Fig. 1).

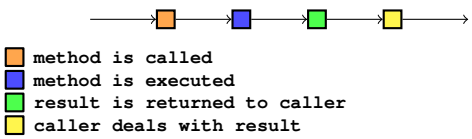

FIGURE 1. call of an object method

With active objects, calling and execution are asynchronous. Method calls are request which are sent to an active object. This last one stores requests in a queue and executes them according to a scheduler (ie Fig. 2).

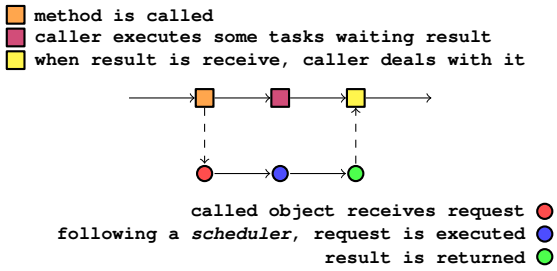

FIGURE 2. call of active object method

Figure 3 shows this process : caller send a request, with the help of a stub object, to an active object. This active object returns a *future* with an empty value. When request has been executed, value of the future is set and caller can handle this value.

Interest of active objects is that caller can execute other tasks while it is waiting for a request's result : call is not blocking anymore. Each active object has its own thread to execute received requests. A *stub object* is used to contact an active object on a method call. This call is done through a proxy which is a bridge between stub objects and the active object.

B. Middleware

A middleware provides a connection between softwares or between components of a software. This connection allows a communication between process. These process can be located

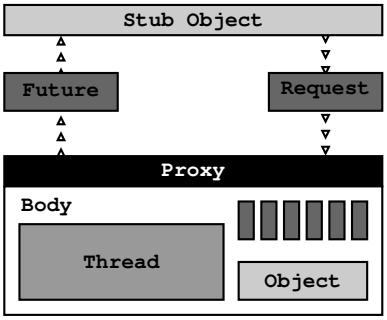

FIGURE 3. Active Object pattern

on different machines, with different operating systems. So, a middleware allows a program to exploit power of several participating computers. It can also manage connection and disconnection of participants. So, it creates a dynamic grid of machines used to distribute tasks.

There is different types of middleware. Some of them provide to developers a way to send a job to a machine and then, when job is done, retrieve the result. In this case, there is no interaction between participants, and it is usually a centralized approach : a server (or a restricted set of servers) distributing job to some slaves. For example BOINC[2], which is used by `@home` projects like `SETI@home`[1].

Others have an object-approach of the distribution using for some of them the active object pattern. It is the case of ProActive[3] developed by INRIA Sophia Antipolis. This kind of middleware allows interactions between different distributed tasks.

With program running on a single machine, there is an addressing space which allows to attribute an unique id to each object. This id is often the memory address of the object. When a program is distributed (it is running on a set of machines), uniqueness of an id is more difficult to assure. Middlewares provide a global addressing space which allow an unique id for each object.

C. Load-balancing

The concept of load-balancing is that a set of machines $\mathbb{S}$ are grouped as an unique virtual machine $\mathbb{M}$. If $\mathbb{T}$ is the set of all executable tasks, then load-balancer is a function $l : \mathbb{T} \rightarrow \mathbb{S}$ which attributes to a task t a machine $l(t) = m$ that will execute the task t. So the load-balancer l defines a policy to distribute tasks on $\mathbb{M}$.

This technique is used by web services, for example, to spread users's requests between servers : several servers look like one for users. When a user sends a request to a web service, load-balancer redirects this request to one of the available servers. This allows to equilibrate work-load of servers and provide a best quality of service.

In distributed programming, it allows to optimize work-load of each machine by establishing a policy to spread tasks on machines. Some load-balancing algorithms can depend on the type of distributed network : this network can be synchronous or asynchronous and its topology can be dynamic.

1. http://setiathome.ssl.berkeley.edu/

D. Dynamic Graph

Execution of simulations distributed by DAGDA can be described as a dynamic graph, so the concept of graph and then of a dynamic graph need to be defined.

A graph G is a pair (N, E) where N is a set of elements called nodes and E is a set of nodes-pair (u, v) called edges such that $u, v \in N$. A dynamic graph is a sequence $G_i = (N_i, E_i)$ such that $\forall (u, v) \in E_i, u, v \in N_i$. So, it is a graph that can change over time, by adding/removing nodes and/or edges, but such that if an edge exists at a time i then its ends also exist at the same time i.

Nodes of the graph are the entities of the simulations and edges model interactions between entities. Edges of the graph modeling the execution are weighted : more intensive is an interaction between two entities, more important is the weight of the corresponding edge.

II. ANTCO^2

ANTCO^2 is a distributed algorithm dedicated to load balancing and communication minimisation.

AntCo^2 considers only the dynamic graph of the application to compute the distribution.

As communications and agents appear and disappear, as the importance of communication evolve, the graph changes. Therefore the load balancer should also handle this dynamic process and be able to provide a distribution as the graph evolve.

Each computing resource is associated with a color, then by assigning a color to a node, the algorithm specify the distribution.

One can see the distribution as a weighted partionning of the graph. In this partionning we try to distribute evenly the load (number of entities weighted by their computing demand) and to minimize communications between computing ressources to avoid saturating the network. These two criteria are conflicting, therefore a trade-off must be found.

We see the partionning as a dynamic community detection algorithm. We call such dynamic communities "organizations". Communities are often seen as group of vertices that are more densely connected one with another than with the rest of the graph. An algorithm able to detect organizations is able to follow communities as they evolve when nodes and edges appear, evolve and disappear in the communities.

There exists several graph partionning algorithms ([4], [5], [6]) and community detection algorithms ([7]), but few handle evolving graphs. It is always possible to restart such algorithms each time the graph changes, but this would be computationally intensive. AntCo^2 is an incremental algorithm that start from the previous partionning to compute a new partionning when the graph changes.

Having a load balancer running on a single machine, to distribute applications that are often very large could be inefficient. Another goal of AntCo^2 is to be able to be distributed with the application.

AntCo^2 uses an approach based on swarm intelligence, namely colonies of ants. This algorithm provides several advantages : ants can act with only local knowledge of the

graph representing the application to distribute. AntCo^2 tries to avoid any global computation, therefore allowing it to be distributed with few communications and no global control.

In AntCo^2, each colony represents a computing resource and has its own color. Inside colonies, ants collaborate to colonize organizations inside the graph and assign their color to nodes. Inversely, colonies compete to keep and conquer organizations.

Ants color nodes using numerical colored pheromones corresponding to their colony color. Such pheromones "evaporate" and therefore must be maintained constantly by ants. This allows to handle graph dynamics by forgetting old partionning solutions and discovering new solutions by the constant exploration of ants inside the graph. The details of the algorithm are given in ([8]).

The change of a color for a node indicates a "migration advice", meaning that the corresponding agent should migrate on the computing resource associated to the new color. An inerty mechanism allows to avoid oscillatory advices.

III. Dagda

DAGDA is a middleware dedicated to the distribution of Complex Systems simulations. It uses an existing middleware as a base which is extended with new features. The final aim is to provide a simple way to create distributed complex system simulation.

The main words of Dagda are decentralized, portable, load-balanced. Decentralized means that there is no restricted set of machines on which depend all machines. Dagda aims to be as portable as possible, is that any machines (computer,pda,phone,super-calculator,...) can participate to the distribution.

The used middleware is ProActive[3]. This choice is motivated by the active object approach which is used in ProActive.

A. Entities

Dagda is based on the concept that the distributed application is composed of a massive set of objects. These objects are called *entities* and are hosted on a machine by an *agency*. Entities are active objects.

Entities can interact with each other and can migrate from one agency to an other. This rises a problem : how to identify each entity through the network and how to get a remote entity ? The second part of this problem, how to get entity, is treated on section III-B. Entities are identified by an id which is unique through time and network. Uniqueness is assumed by the fact that id depends on the agency's address (agency who creates the entity) and on a timestamp.

B. Communication between agencies

Dagda aims to have a decentralized architecture so there is no master server to reference informations as for example entities location. Therefore, some mechanisms are needed to provide functionalities like entity-search.

Each agency has a dedicated active object whose role is to detect other agencies and to provide to user functionalities through connected agencies.

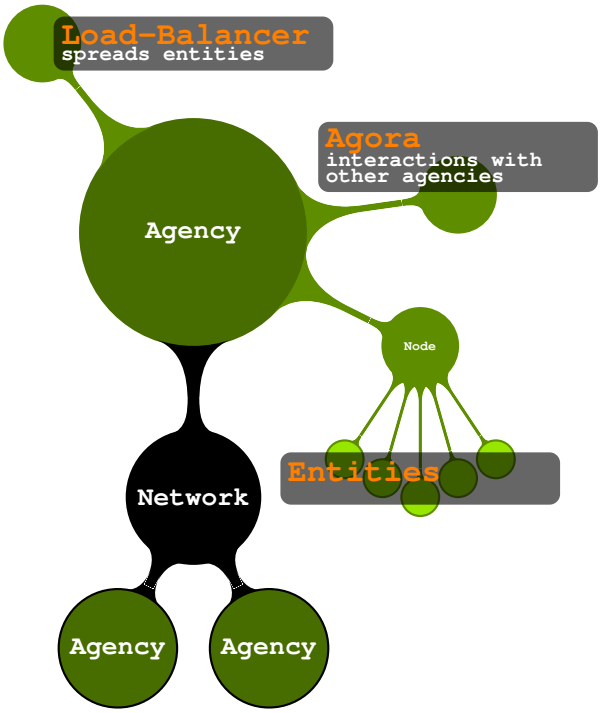

FIGURE 4. Dagda overview

C. Context

A program may have some parameters which create a context that is used by composants of this program. With programs running on a simple process, this is easy to do by declaring global variables. But with distributed programs, each machine has its own memory and one other can not see changes.

Dagda creates a context divided in two parts. It contains a *local* part which contains parameters that do not have to be shared. Second part is *global* and changes on this part will be spread on all machines. Context users can access to parameters without local or global distinctions.

D. Interactions Graph

Dagda profiles method calls between entities. For example, if an entity `A` calls a method `m()` of an entity `B`, this call will be detected and registered. Then this detection of interactions between entities is used as provider to a dynamic graph which models these interactions through the time. Nodes of this graph are the entities host on the machine and remote entities such that there is an interaction between these remote entities and one of the hosted entities. Edges of the graph represent interactions between entities. Greater is the number of interactions between two entities, greater is the weight of the corresponding edge. There is a mechanism which decreases edges's weight through the time. The GRAPHSTREAM[9] [2] API is used to create the graph.

This graph can be used by tools, for example to monitor entities activity and have a look on this activity.

2. http://www.graphstream-project.org

E. Load-balancing

Entities are spread on the available machines with the ANTCO^2 load-balancing algorithm. This choice allows :

- equilibrate the work-load of machines ;
- reduce the network-load ;
- distribute the load-balancer.

Distribution of the load-balancer is an important thing to have a decentralized platform. In [10], three ways are presented to run the ANTCO^2 algorithm. The first and the second one run with a restricted set of servers (size of this set is one in the first case). In these two cases, computing-load of server is fully used and ANTCO^2 has a global view of the distributed application. The last one uses each machines to run the algorithm. In this case, only a few computing-load is used on each machine and ANTCO^2 has just a local view of the distributed application. This case allows to decentralize ANTCO^2, so it has been choosen in DAGDA.

Work-load dedicated to ANTCO^2 is function of number of entities, so by balancing entities-load, it balances itself : it is auto-distributed.

IV. RESULTS

AT this time, DAGDA is still in development. The platform is able to create entities and profile interactions between them. So it is possible to view the graph of the simulation's execution in real time. It is also able to connect agencies and migrate entities from one agency to another.

A. Test application

To realize some tests, a simple application has been written which aimed to generate interactions between entities and migrate them between agencies. Entities used for this application can be described as follows :

```
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();
    }
```

The application creates a set of `TestEntity` and inits randomly the *neigh* attribute of entities. Then each agency run the *execute()* method of each hosted entity.

Figure 5 shows the graph of the execution of this application with 64 entities.

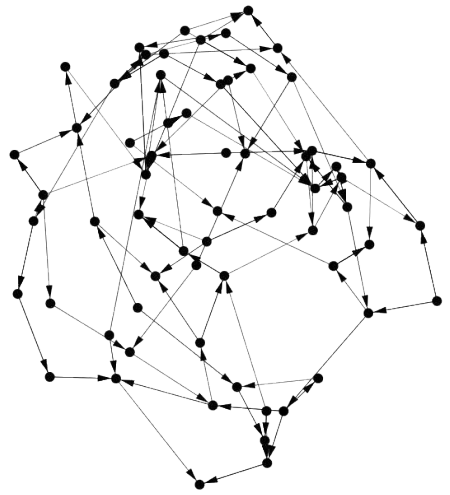

FIGURE 5. Execution of the test program

V. CONCLUSION

In this paper, concepts of middleware and load-balancing have been described. Then the DAGDA platform which merges a middleware and the load-balancer ANTCO^2 has been presented.

DAGDA is still in development but it ables to launch a program and profile execution of this program. Next step is to finalize implementation of the load-balancer and validate the platform making battery of tests.

Then we need to provide an application running on DAGDA and realize tests on performances to show the gain brought by DAGDA.

RÉFÉRENCES

[1] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *IJCAI*, 1973, pp. 235–245.

[2] D. P. Anderson, "Public computing : Reconnecting people to science," in *Conference on Shared Knowledge and the Web*, Residencia de Estudiantes, Madrid, Spain, Nov. 2003.

[3] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici, *Grid Computing : Software Environments and Tools*. Springer-Verlag, January 2006, ch. Programming, Deploying, Composing, for the Grid.

[4] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graph," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 192–307, 1970.

[5] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," *SIAM J. Scien. Comput.*, vol. 16, no. 2, pp. 452–469, 1995.

[6] C. M. Fiduccia and R. M. Mattheyses, "A linear time heuristic for improving network partitions," in *ACM IEEE Design Automation Conference*, 1982, pp. 175–181.

[7] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev*, vol. 69, 2004.

[8] C. Bertelle, A. Dutot, F. Guinand, and D. Olivier, "Organization detection for dynamic load balancing in individual-based simulations," *Multi-Agent and Grid Systems*, vol. 3, no. 1, p. 42, 2007. [Online]. Available : http://litis.univ-lehavre.fr/~dutot/biblio/MAGS2007.pdf

[9] A. Dutot, F. Guinand, D. Olivier, and Y. 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.

[10] A. Dutot, “Distribution dynamique adaptative à l’aide de mécanismes d’intelligence collective,” Ph.D. dissertation, Université du Havre - LIH, 2005.

[11] M. K. et al., “Weighted round-robin cell multiplexing in a general-purpose atm switch chip,” *IEEE J. of Selected Areas in Comm.*, vol. 9, no. 8, pp. 1265–1279, 1991.