RAGE: an agent framework for easy distributed computing

P. Mathieu, J.C. Routier, and Y. Secq

Laboratoire d'Informatique Fondamentale de Lille – CNRS UPRESA 8022
UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
59657 Villeneuve d'Ascq Cedex
{mathieu,routier,secq}@lifl.fr

Abstract. This paper presents the RAGE framework. RAGE stands for *Reckoner AGEnts*. It is an agent framework for the *easy* design of distributed computing environment. RAGE has been developped using the MAGIQUE multi-agent framework. Because distribution, cooperation and organizations are key concepts in multi-agent systems, they are natural solutions for the design of such applications. The use of MAGIQUE has contributed to an easy development of RAGE and provides to the application the capacity to easily evolve and adapt. The goal of this article is not only to present RAGE but also to show how the development and evolutions of such an application with a multi-agent system is simple.

1 Introduction

One application field of multi-agent is *Distributed Problem Solving*, and one typical example of such problems is the distributed calculus. Because distribution, cooperation and organizations are key concepts in multi-agent systems, they are natural solutions for the design of applications of distributed computing.

The use of a multi-agent framework allows the designer to get rid of the problems that would have been generated by the distribution and the communications between (what would have been) clients. Indeed these are primitives in multi-agent infrastructures and are often even hidden to the designer. Moreover, the agent approach (or paradigm) leads the designer to a natural decomposition of the problem in term of "tasks" and roles. The notion of *role* has been largely emphathized in works related to actor languages [6], subject oriented programming [5], and of course in the multi-agent field [3]. This notion relies on the specification of the behaviour of an actor/agent. In a sense, role can be seen as an equivalent of interfaces (or pure abstract classes) in object oriented programming. The main difference resides in the fact that interactions are not constrained with interfaces (anybody can invoke a method), while with roles one can ensure that the request comes from the right role. The other point of roles is that they identify the required functional requirements and are not tied to particular agents.

For all these reasons we claim that multi-agent is an appropriate framework for development of easy to use, ease to evolve, distributed calculus applications.

Clustering computers to share their power and exploit their idleness is an idea that has gain momentum with the advent of Internet. Projects like distributed.net[1] or SETI@Home [2] illustrate this trend. Despite their interest, they are centralized (they rely on a client-server approach) and they are monolithic based frameworks (clients are mono-calculus). RAGE instead proposes a framework where clients does not only provide computing power but can also create their own calculus. This approach is closer to frameworks like xDu[4] or JavaParty[7].

In the first part we present, briefly, the MAGIQUE framework. It is based on an organisational model and on an agent model which have been put into concrete form as a JAVA API. MAGIQUE has been used to build the *easy* distributed computing environment application described in the second part: RAGE. We will describe both how this application has been designed and the resulting framework.

2 Magique : a Multi-Agent framework

MAGIQUE proposes both an organisational model [1], based on a default hierarchical organisation, and an agent model [9], which is based on an incremental building of agents.

The agent model: building agents by making them skilled. The agent model is based on an incremental building of agents from an elementary (or atomic) agent through dynamical skill acquisition. A skill is a "coherent set of abilities". We use this term rather than service[2], but you can consider both as synonyms here. From a more programmer oriented view, a skill can be seen as a software component that groups a coherent set of functionalities. The skills can then be built independently from any agent and reused in different contexts.

We assert that only two prerequisite skills are necessary and sufficient to the *atomic agent* to evolve and reach any wished agent: one to interact and one to acquire new skills [3].

Thus we can consider that all agents are at birth (or creation) similar (from a skill point of view): an empty shell with only the two above previously mentioned skills.

Therefore differences between agents are issued from their "education", i.e. the skills they have acquired during their "existence". These skills can either have been given during agent creation by the programmer, or have been dynamically learned through interactions with other agents (now if we consider the programmer as an agent, the first case is included in the second one). This approach does not introduce any limitations to the abilities of an agent. Teaching skills to an agent is giving him the possibility to play a particular role into the multi-agent system he belongs to.

---

[1] http://www.distributed.net/
[2] We keep *service* for "the result of the exploitation of a skill".
[3] Details can be found in [9].

This paradigm of dynamic construction of agent from skills, has several advantages from a programming point of view: the *development becomes easier* (modularity given by the skills), *efficiency* (skills distribution can be dynamilly adapted), *robustness* (critical skill can be preserved), *autonomy and evolutivity*, *runtime customization* (adaptation to runtime environment).

The organisational model. In Magique, there exists a basic default organisational structure which is a hierarchy. It offers the opportunity to have a default automatic mechanism to find a skill provider.

The hierarchy characterizes the basic structure of acquaintances in the multi-agent system and provides a default support for the routing of messages between agents. A hierarchical link denotes a communication channel between the implied agents. When two agents of a same structure are exchanging a message, by default it goes through the tree structure.

With only hierarchical communication, the organisation would be too rigid and Magique offers the possibility to create direct links (i.e. outside the hierarchy structure) between agents. We call them *"acquaintance links"* (by opposition of the default *hierarchical links*). The decision to create such links depends on some agent policy. However the intended goal is the following: after some times, if some request for a skill occurs frequently between two agents, the agent can take the decision to dynamically create an acquaintance link for that skill. The aim is of course to promote the "natural" interactions between agents at the expense of the hierarchical ones.

With the default acquaintance structure, an automatic mechanism for the delegation of request between agents is provided. When an agent wants to exploit some skill it does not matter if he knows it or not. In both cases the way he invokes the skills is the same. If the realization of a skill must be delegate to another, this is done transparently for him, even if he does not have a peculiar acquaintance for it. The principle of the skill provider search is the following:

- the agent knows the skill, he uses it directly
- if he does not, several cases can happen
  - first he has a particular acquaintance for this skill, this acquaintance is used to achieve the skill (ie. to provide service) for him,
  - he is a supervisor and someone in his hierarchy knows the skill, then he forwards (recursively through the hierarchy) the realisation to the skilled agent,
  - he asks its supervisor to find for him some gifted agent and his supervisor applies the same delegation scheme.

One first advantage of this mechanism of skill achievement delegation is to increase the reliability of the multi-agent system: the particular agent who will perform the skill has no importance for the "caller", therefore he can change between two invocations of the same skill (because the first had disappeared of the multi-agent system or is overloaded, or ...).

Another advantage appears at the programming stage. Since the search of a skilled agent is automatically achieved by the hierarchy, when a request for a skill is programmed, there is no need to specify a particular agent. Consequently the same agent can be used in different contexts (i.e. different multi-agent applications) so long as an able agent (no matter which particular one) is present. A consequence is, that when designing a multi-agent system, the important point is not necessarily the agents themselves but their skills (ie. their roles).

The API These models have been put into concrete form as a JAVA API, called MAGIQUE too. It allows to develop multi-agent systems distributed over heterogeneous network. Agents are developed from incremental (and dynamical if needed) skill plugging and multi-agent system are hierarchically organized. As described above, some tools to promote dynamicity in the multi-agent system are provided: direct acquaintance links can be created, new skills can be learned or exchanged between agents (with no prior hypothesis about where the byte-code is located, when needed it is transferred between agents). The API can be downloaded at `http://www.lifl.fr/MAGIQUE`.

3 Rage: an easy distributed computing framework

In this section, we will illustrate how we have designed and implemented RAGE using the MAGIQUE infrastructure. We will see that an agent oriented approach offers simplicity in the development of such an application, and promotes an easy evolution of the system too.

We will briefly present the framework, that is the point of view of the framework user, then we will see how it has been designed, which is the point of view of the designer. While the end user can see only distributed entities, he can call them agents or distributed objects if he prefers, the designer clearly tackle the problem using agent notions: agents, interactions and organisation.

3.1 The *Rage* framework: the end user point of view

RAGE[4] wants to be a framework that offers an easy development of distributed calculus and allow multi-applications computing in parallel and scaling in the available power (cluster).

Simplicity was a preliminary condition for the framework. The main idea was to provide an easy framework for non computer scientists. To achieve this goal, we had to defined a small number of concepts that the user has to understand in order to feed the system with its calculus. It is important to note that the end user has no need to know anything about agent or multi-agent systems, even if he has to understand at least some object-oriented notions (since he must inherits some predefined patterns). The user has mainly to know two concepts: what is a *task* and what is a a *result*. A *task* can be seen as the algorithm that is

---

[4] for Reckoner AGENT

distributed, while a *result* represents the datas produced by the *task* and which must be stored.

The user of the framework has to define what should be distributed, and what is the global scheduling of its main algorithm. The *task* defines the chunk of algorithm that will be distributed among agents. The simplest way for the user to create a *task* is to subclass the `AbstractTask` class and to define the only two methods:

```
abstract public void compute();
abstract public boolean finished();
```

The complexity for the user is then not bigger than writing a Java Applet. Therefore, the end user has a rather OO view of the framework: he extends one class to adapt it to his distributed application and uses the underlying framework without necessary explicitly knowing what happens.

3.2   Implementation with Magique: the designer point of view

The design of the framework has loosely followed the GAIA methodology [10] and has been defined through those steps: first, definition of the roles involved in the framework and their functionnalities (or skills), second, definition of the organisation of roles in the system and last mapping of the roles on agents.

The first step identifies the main entities of the application, and the functionnalities they have to assume. This step describes also the interactions between the roles. The second step defines the number of agents playing a particular role and their position in the organisation. The last step do the mapping between the agents and the roles they play.

The first task is to define the roles. A short analysis of the problem allows to determine a set of roles that can be distinguished to bring a solution to the problem of distributed calculus. Here is a short description of these roles:

Boss role it is responsible of all the interactions with the user part

Task Dispatcher role it has to dispatch *tasks* and deal with the fault tolerancy.

Platform Manager role it manages the agents that will compute the *tasks*, and must ensure that there are always available *tasks* for these agents.

Reckoner Agent role it is the worker of the framework, it computes *tasks*, and send back the *results* of this computation.

Repositories Manager role it manages the storage of *results*

Result Repository role it is a mirror of the database of *results*

As we implemented the system with Magique, we have put in concrete form those roles with the corresponding skills. For example, the *Platform Manager role* is defined by a Magique skill that implements its goal: getting tasks and dispatching them to *Reckoner Agents*. A role is then defined by a set of skills that forms its functionnalities. The interaction between the roles must then be described. With Magique, the dynamic of interactions between roles is contained in the skills.

Now that we have seen the involved entities, let us have a look on their interactions, that is in the backstage of the application. While the user "runs" its calculus, he first sends his *tasks* to the framework, then they are stored until an agent requests them. Once the agent has computed its *task*, he sends the *result* of the computation back to a repository. Thus the user can retrieve them and go on with its main algorithm. For the user, distribution and computation are totally abstract, he only has to feed the framework with *tasks* and retrieves the *results*, the agents of the framework manages everything for him.

This stage can be considered as the analysis phase. We have to define the functionnalities associated to a role. This lead to the definition of an interface of skill. It is important to work on interfaces, as skill implementations are not considered at this stage. Actually, the implementation will be dependant of the available runtime. We will indeed not have the same skill running on a workstation and on a cellular phone, but the interface of the skill will be the same.

Once those entities are identified, the architecture of the system must be defined. When working with MAGIQUE, it means that the architecture is shaped by the logical grouping of roles and by the dynamic of interactions. If we take a look back at interactions: users send *tasks* to the *Boss*, which sends them to the *TaskDispatcher*. Then, *PlatformManagers* that are available request *tasks* and dispatch them to available *ReckonerAgents*. Once the computation of a *task* is done, the *result* is sent to the *Repositories Manager*. The figure 1 shows the
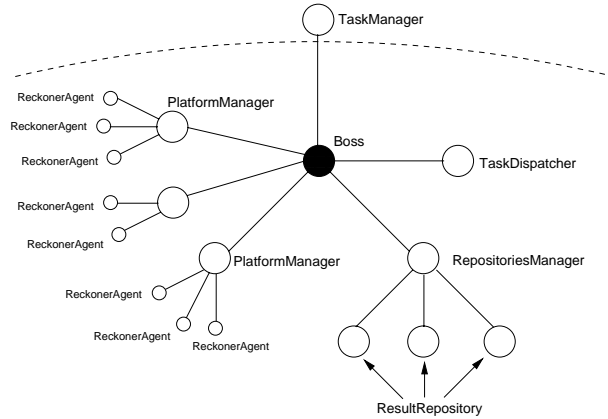

Fig. 1. Rage's organisation: the hierarchy of roles.

logical hierarchy of roles, it does not define how roles are mapped onto agents and how agents are mapped onto the network of computer, but instead provides the topology of what we could call *natural acquaintances*. There is no a priori reason to respect a "one role–one agent" rule, the whole hierarchy could even be mapped onto one single agent (that would not be useful, but could be done!).

Here we have quickly seen the designer vision: the determination of roles and their interactions and then the choice of the organisation. The agent oriented approach has allowed a quick development of the application. Moreover, we see that even if finally the user has not necessarily an agent vision of the application, the design was agent oriented.

4   Conclusion

The presented application, RAGE, is an easy to use framework for distributed computing. It allows to develop, distribute and run calculus on heterogeneous framework with no need of background in distributed computing or agents technologies. The user can only direct its efforts towards his very calculus.

This article argues that multi-agent paradigms: agent animacy, organisation and roles, are key notions to support the analysis, design and implementation of open large-scale distributed systems. It is particulary significant that those application models are in adequation with multi-agent paradigms. It has been illustrated with an application of distributed computing, but could be extended to other classes of applications (like CSCW[8]).

From our point of view, RAGE demonstrates that agent oriented programming is an appropriate framework for the development of such a distributed applications. And as a consequence, the resulting framework is easy, first, for the user whos has a calculus to do, and, second, for the designer who wants to extend the capacities of the framework.

References

1. N.E. Bensaid and P. Mathieu. A hybrid and hierarchical multi-agent architecture model. In *Proceedings of PAAM'97*, pages 145–155, 1997.
2. Anderson et al. Seti@home: The search for extraterrestrial intelligence. Technical report, Space Sciences Laboratory, University of California at Berkeley., 1999.
3. J. Ferber and O. Gutknecht. Operational semantics of a role-based agent architecture. In *Proceedings of ATAL'99*, jan 1999.
4. Samir Gehani Gregory. xdu: A java-based framework for distributed programming and application interoperability.
5. W. Harrison. Subject-oriented programming, 1993.
6. C. Hewitt. Viewing control structures as patterns of passing messages. In *Artificial Intelligence: An MIT Perspective*. MIT Press, Cambridge, Massachusetts, 1979.
7. Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.
8. JC. Routier and P. Mathieu. Une contribution du multi-agent aux applications de travail coopératif. *TSI Hermès Science Publication*, Numéro Spécial : Téléapplications, To appear: 2001.
9. JC. Routier, P. Mathieu, and Y. Secq. Dynamic skill learning: A support to agent evolution. In *Proceedings of the AISB'01 Symposium on Adaptive Agents and Multi-Agent Systems*, pages 25–32, 2001.
10. M. Wooldridge, NR. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 2000.