AISB Journal **1**(2), © SSAISB, 2002
http://www.aisb.org.uk

Using agents to build a distributed calculus framework

Philippe Mathieu, Jean-Christophe Routier, Yann Secq

Equipe Systèmes Multi-Agents et Coopération
Laboratoire d'Informatique Fondamentale de Lille
CNRS UPRESA 8022
Université des Sciences et Technologies de Lille
{mathieu, routier, secq}@lifl.fr

Abstract

This article argues that multi-agent systems can be seen as an interesting paradigm for the design and implementation of large scale distributed applications. Agents, interactions and organisations, that are the core of multi-agent systems, consitute a well fitted support to the design of complex real-world software. They combine the well defined and desired properties for good software engineering: decomposition, abstraction and organisation.

This paper presents an agent framework RAGE, *Reckoner AGEnts*, for an *easier* design of distributed calculus applications. This framework has been developped using the MAGIQUE multi-agent framework. Because distribution, cooperation and organization 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 easier development of RAGE and provides to the application the capacity to smoothly evolve and adapt.

In the first part of this paper, we introduce concepts that are emphasized in the multi-agent domain. Then, we present the principles of the MAGIQUE framework. The second part describes RAGE framework and details a practical example. Finally, the last part illustrates how an agent oriented approach has been followed for the design and implementation of RAGE.

1 Introduction

One application field of multi-agent systems (MAS) is *Distributed Problem Solving*, and one typical example of such problems is the distributed calculus. Because distribution, cooperation and organizations are key concepts in MAS, 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 *agents*, *tasks* and *roles*. We will not give here another definition of "what is an agent"[1], but we will stick to the one proposed in (Wooldridge, 1997): *an agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives*. The notion of *role* has been largely emphathized in works related to actor languages (Hewitt, 1979), subject oriented programming (Harrison, 1993), and of course in the multi-agent field (Ferber and Gutknecht, 1999). This notion relies on the specification of the behaviour of an actor/agent. In a sense, roles 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 with *roles* is that they identify the functional requirements and are not tied to particular agents.

For all these reasons we claim that multi-agent systems are appropriate infrastructures for the design and development of flexible frameworks for 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 (Gray and Sunderam, 1997; Stankovic and Zhang, 1998). Projects like distributed.net[2] or SETI@Home (Anderson and al., 1999) 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(Gregory, 1998) or JavaParty (Philippsen and Zenger, 1997).

In the first part of this paper we present, briefly, the MAGIQUE framework. It is based on an organisational model and on an minimal 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 agent model(Routier et al., 2001), which is based on an incremental building of agents, and an organizational model (Bensaid and

[1] For a lot of definitions of "agent", have a look on (Franklin and Grasser, 1996)

[2] `http://www.distributed.net/`

Mathieu, 1997), based on a default hierarchical organization.

2.1 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[3], but you can consider both as synonyms here. From a developer point view, a skill can be seen as a software component that gathers 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(Routier et al., 2001).

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 developer, 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 MAS he belongs to.

This paradigm of dynamic construction of agent from skills, has several advantages :

- *development becomes easier* : modularity is given by the skills,
- *efficiency* : skills distribution can be dynamilly adapted,
- *robustness* : critical skill can be preserved,
- *autonomy and evolutivity* : runtime customization available through adaptation to runtime environment.

2.2 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 delegation mechanism to find a skill provider.

The hierarchy characterizes the basic structure of acquaintances in the MAS and provides a default support for the routing of messages between agents. A hierarchical link denotes a communication channel between the implied agents.

[3]We keep *service* for “the result of the exploitation of a skill”.

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, thus 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: 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
    - if he has a particular acquaintance for this skill, this acquaintance is used to achieve the skill (ie. to provide service) for him,
    - else he is a supervisor and someone in his sub-hierarchy knows the skill, then he forwards (recursively through the sub-hierarchy) the realisation to the skilled agent,
    - else he asks its supervisor to find for him some skilled 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 MAS or is overloaded, or ...).

Another advantage appears while developping applications. 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).

2.3 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 MAS are provided: direct acquaintance links can be created, new skills can be learned or exchanged between agents (with no prior hypothesis about where the bytecode is located, when needed it is transferred between agents). The API, a tutorial and samples applications 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 name them agents or distributed objects if he prefers, the designer clearly tackles the problem using agent notions: agents, interactions and organization.

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, that allows multi-applications computing in parallel and that can scale with 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 define 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 distributed, while a *result* represents the data 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. Let us take the example of a matrix that has to be computed and then used to solve a linear system. The main algorithm could be cut in two phases: first compute the matrix (distributed), then

[4] RAGE stands for Reckoner AGEnt

solve the system (centralized). In the first stage the user dispatches the tasks that populate the matrix, then he retrieves the matrix to solve the system. 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 tailor it to his distributed application and uses the underlying framework without necessary explicitly knowing what happens.

To ease the migration or cancellation of running tasks, the user has to follow a simple principle while designing the `compute()` method : tasks are considered as cooperative in respect to the threading model[5]. The figure 1 illustrates this principle and describe the lifecycle of a `Task`.

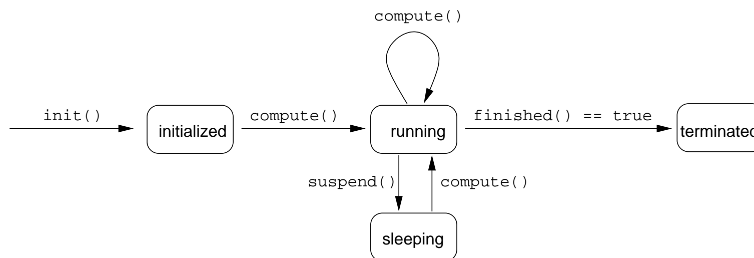

Figure 1: The lifecycle of a RAGE *Task*.

To illustrate what the user must do, we will detail a simple example : computing an approximation of the π number. The idea of the algorithm is to randomly choose points in an unitary square, and to check wether the point is within the quarter of circle or not. Then, we can apply the following formulae : $\pi = 4 * I/N$, where N is the total number of points and I is the number of points within the quarter of circle (see figure 2).

The accuracy of this algorithm increases when N grows. Thus, the user will have to define a `PiTask` that encapsulates this algorithm, to create a set of these tasks and to dispatch them within RAGE. The complete source code for the `PiTask` class is given here.

```
public class PiTask extends AbstractTask {
  public Double pi;
  private int crtIter = 0, inner = 0, niter, chunk;
```

[5]This model was inspired by works done on the FAIRTHREADS project at `http://www-sop.inria.fr/mimosa/rp/FairThreads/`

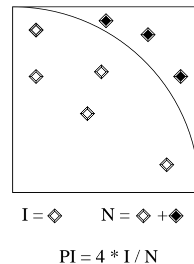

Figure 2: A Monte-Carlo algorithm to compute an approximation of π

```
  public PiTask(String factId, String taskId, int niter){
    super(factId, taskId);
    this.niter = niter; chunk = niter/10;
  }
  public void compute(){
    double x, y;
    for(; (crtIter % chunk) < chunk; crtIter++){
      x = Math.random(); y = Math.random();
      if (Math.sqrt(x*x + y*y) <= 1.0)
        inner ++;
    }
    pi = new Double(4*((double)inner/(double)niter));
  }
  public boolean finished(){
    return crtIter == niter;
  }
  public double percent(){ // Percent of progress
    return (double) (crtIter * 100)/(double)niter;
  }
}
```

Despite its simplicity, this example illustrates the programming model of the RAGE framework :

- extend `AbstractTask` (or implement the `Task` interface),
- implement the `compute()` and `finished()` methods,
- tag as `public` members that should be saved as result.

Then, the user creates a bunch of tasks and sends them to the framework. Later, he can retrieve the results and compute the average value. One interesting point of the `compute` method is that it illustrates the delegation of control from the task to the agent that handles it. If the agent receives an order to terminate or migrate the task, it can be handled only if the user releases the control to the agent. Thus, a `PiTask` will need 10 calls to the `compute` method before being achieved. The granularity of the `compute` method should therefore be carefully studied by the user if he wants to take advantages of task migration or termination.

3.2 Implementation with MAGIQUE: the designer point of view

The design of the framework has loosely followed the GAIA methodology (Wooldridge et al., 2000) and has been defined through those steps:

- firstly, the definition of the roles involved in the framework and their abilities (or skills)
- secondly, the definition of the organization of roles within the system
- and finally, the mapping of roles on agents.

The first step identifies the main entities of the application, and the abilities they have to assume. This step describes also the interactions between roles. The second step defines the number of agents playing a particular role and their position in the organization. Then, the last step do the mapping between the agents and the role(s) 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 is responsible of all the interactions with the user part

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

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

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

Repositories Manager role manages the storage of *results*.

Result Repository role 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* : agents of the framework manage everything for him.

This stage can be considered as the analysis phase. We have to define the abilities associated to a role. This leads to the definition of a skill interface. 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 3 shows the logical hier-

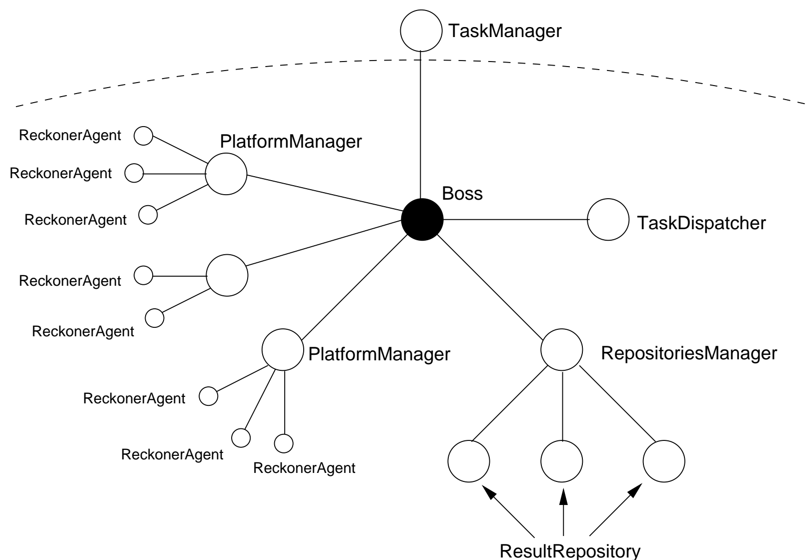

Figure 3: Rage's organization: the hierarchy of roles.

archy 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 call *natural acquaintances* structure : the default organization.

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 it could be done!). It is important to note that we could have applied the same role decomposition with another agent infrastructure, and then used another topology of organisation. If we had used Madkit (Gutknecht O., 2000) for example,

the organisation would have followed the Aalaadin model (Ferber and Gutknecht, 1998), and thus we whould have had organize roles with groups instead of the hierarchy.

Here we have briefly seen the designer vision: the determination of roles and their interactions and then the choice of the organization. The agent oriented approach has allowed a quick development of the RAGE framework. Moreover, we see that even if finally the user has not necessarily an agent vision of the framework, the design was agent oriented. And while the framework is running, agents are working on behalf of RAGE users.

3.3 Experiments done with RAGE

To evaluate the framework, we have done some experiments that are ranging from simple examples to complex applications. The first experiment is a *tutorial* example that we have seen in section 3.1 (i.e. computation of an approximation of π). It is based on a Monte-Carlo method and illustrates how simple it is to create a *task* and to run the framework.

The second tutorial experiment is a *naive* implementation for prime number decomposition, which enabled us to evaluate the scaling of the infrastructure. The idea is to test the primality of a number by making an exhaustive search of its factors.

The third example is bigger : it is an exploration of the underlying structure of the Donkey sliding block game. The algorithm consists in an exhaustive generation of game states graphs(Elwyn R. Berlekamp, 1982). It is interesting as it works in two stages and shows how computations can be canceled. The first stage consists in the creation of all valid states of the game. Then, we choose randomly a state that will be considered as a seed to create the graph of all accessible states from it. This is where the cancellation of computation is needed : several seeds are chosen and computations take place, but sometimes we have to check that we are not computing the same graph from different seeds. Thus, when a graph G is complete, each task receives the set of states of G and can therefore check if they are computing the same graph or not.

The last example is an application for solid mechanics : it is an implementation of the two-dimensional displacement discontinuity method (Crouch and Starfield, 1983). This sample provides a way to introduce the idea of shared data because each task needed to access a matrix representing forces. This mecanism of shared data is handled by *PlatformManager*s : they retrieve data from the user and make them available to their *ReckonerAgent*s.

These experiments along with the framework can be downloaded at `http://www.lifl.fr/SMAC/projects/magique/examples`.

3.4 What is next on the framework ?

The RAGE framework misses some features that could ease its use and broaden the field of applications it can handle. In this section, we present the most needed enhancements, firstly within the framework himself, and secondly around it.

The main missing feature in RAGE is the lack of inter-tasks communication. It has not yet been implemented, but thanks to the underlying multi-agent system, it can be quite easily handled. As tasks are managed by *ReckonerAgent*s, inter-tasks communication could be provided by using communication paths provided by the hierarchy. It would involve the implementation of a search algorithm between *PlatformManager*s : if task identification is done through their identifier, the implementation is straightforward, but not really usable. It would be better that each task provides some meta-description, and that the search algorithm works on these description to do the matching.

Another improvement that is not yet implemented, but could easily be added is multi-application scheduling. Several computations can be done simultaneously within RAGE, and now the *TaskDispatcher* distributes tasks without any consideration on them. We could add some scheduling policies, thus it could be possible to distribute twice much tasks from one application than another for example. Or, we could even introduce some kind of economical policy(Wolski et al., 2000) where users that provide more computation power to the framework by running *PlatformManager*s, could have a greater priority.

A last important feature that should be implemented concerns the database of results. At present, we are using an object database to ease the deployment of the framework, and there is only one database of results, which is managed by a *ResultRepository* agent. As results can be queried by users and also by tasks that are being computed, we plan to implement some automated mirroring of the database of results. The hardest part will be to establish metrics that could provide some criteria to the framework about the usage of the database. But if we have these criteria, mirroring is easily implemented by dynamical skill acquisition : a new agent is created where needed, the *ResultRepository* role is given to him and then we need to synchronize the local database with others. Another approach could be to implement an automatic mirroring of the database by using some kind of replication mecanisms based on peer-to-peer interactions like the Freenet system(Clarke et al., 2000). The main advantage of this approach is that no metrics have to be defined, replication is only dependent on the usage of data.

Finally, we plan other enhancements around the framework : mainly on tools that can be provided and on packaging. At present, we have mainly worked on the framework, and thus we do not provide any tools to help users to visualize the dynamic of RAGE. An administration tool, that could provide feedback to the user and the ability to dynamically manage tasks and results, would be really useful. The packaging of the framework should be enhanced : we could provide one bundle for the framework, another for computational clients (basically, a *PlatformManager* and its sub-hierarchy). And, it could be interesting to deliver this

client bundle with the JavaWebStart[6] technology, so new clients could easily join a running framework through their browsers.

4 Conclusion

The presented application, RAGE, is an easy to use framework for distributed computing intended to non-compter scientists. It allows to develop, distribute and run calculus over an heterogeneous framework with no need of background in distributed computing or agent oriented technologies. The user can only direct his efforts toward his very calculus.

This article argues that multi-agent paradigms: agent animacy, organization and roles, are key notions to support the analysis, design and implementation of open large-scale distributed systems. It is particulary significant that these application models are in adequation with multi-agent paradigms. It has been illustrated with an application of distributed calculus, but could be extended to other classes of applications like Computer Supported Collaborative Work(Routier and Mathieu, 2002)).

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

References

Anderson and al. (1999). Seti@home: The search for extraterrestrial intelligence. Technical report, Space Sciences Laboratory, University of California at Berkeley.

Bensaid, N. and Mathieu, P. (1997). A hybrid and hierarchical multi-agent architecture model. In *Proceedings of PAAM'97*, pages 145–155.

Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. (2000). Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66.

Crouch, S. and Starfield (1983). *Boundary Element Methods in solid mechanics*. Georges Allen & Unwin.

Elwyn R. Berlekamp, John H. Conway, R. K. G. (1982). *Winning Ways for your mathematical plays*. Academic Press.

[6] http://www.javasoft.com/products/javawebstart

Ferber, J. and Gutknecht, O. (1998). A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings of ICMAS'98*.

Ferber, J. and Gutknecht, O. (1999). Operational semantics of a role-based agent architecture. In *Proceedings of ATAL'99*.

Franklin, S. and Grasser, A. (1996). Is it an agent, or just a program?: A taxonomy for autonomous agent. In *Proceedings of the 3rd International Workshop on Agent Theories, Architectures, and Languages*. Springer-Verlag.

Gray, P. A. and Sunderam, V. S. (1997). IceT: distributed computing and Java. *Concurrency: Practice and Experience*, 9(11):1161–1167.

Gregory, S. G. (1998). xdu: A java-based framework for distributed programming and application interoperability.

Gutknecht O., Ferber J., M. F. (2000). The madkit agent platform architecture. Technical report. `http://www.madkit.org/papers/rr000xx.pdf`.

Harrison, W. (1993). Subject-oriented programming.

Hewitt, C. (1979). Viewing control structures as patterns of passing messages. In *Artificial Intelligence: An MIT Perspective*. MIT Press, Cambridge, Massachusetts.

Philippsen, M. and Zenger, M. (1997). JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242.

Routier, J. and Mathieu, P. (to appear in April 2002). A multi-agent approach to co-operative work. In *Proceedings of the CADUI'02 Conference*.

Routier, J., Mathieu, P., and Secq, Y. (2001). 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.

Stankovic, N. and Zhang, K. (1998). Java and network parallel processing. In *PVM/MPI*, pages 239–246.

Wolski, R., Plank, J., Brevik, J., and Bryan, T. (2000). Analyzing market-based resource allocation strategies for the computational grid.

Wooldridge, M. (1997). *Handbook of Agent Technology*, chapter Agent-Based Software Engineering. IEE Proc. on Software Engineering.

Wooldridge, M., Jennings, N., and Kinny, D. (2000). The gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*.