# JADE-Based A-Team Environment

Piotr Jędrzejowicz and Izabela Wierzbowska

Department of Information Systems, Gdynia Maritime University
Morska 83, 81-225 Gdynia, Poland
{pj, iza}@am.gdynia.pl

**Abstract.** The paper proposes a JADE-based A-Team environment (JADE-A-Team) as a middleware supporting the construction of the dedicated A-Team architectures used for solving variety of computationally hard optimization problems. The paper includes a general overview of the functionality and structure of the proposed environment and a more detailed description of optimization agents including their standard functions, ontology, construction requirements and activation procedure. Further sections explain how to create and activate an A-Team agent and how the communication between agents is handled. Conclusions focus on advantages of the JADE-A-Team environment and on suggestions for further research.

## 1 Introduction

Recently, a number of agent-based approaches have been proposed to solve different types of optimization problems [1], [4], [5]. One of the successful approaches to agent-based optimization is the concept of A-Teams. An A-Team is composed of simple agents that demonstrate complex collective behavior.

The A-Team architecture was originally proposed by Talukdar [9] as a set of objects including multiple agents and memories which through interactions produce solutions of optimization problems. The advantage of the A-Team architecture is that it combines a population of solutions with domain specific algorithms and limited agent interaction. A sophisticated A-Team architecture was proposed in [7]. Some dedicated A-Teams were proposed in [6], [8]. According to [9] an A-Team is a problem solving architecture in which the agents are autonomous and co-operate by modifying one another's trial solutions.

In this paper we propose a JADE-based A-Team environment (in short: JADE-A-Team) as a middleware supporting the construction of the dedicated A-Team architectures used for solving variety of computationally hard optimization problems. JADE is an enabling technology, for the development and run-time execution of peer-to-peer applications which are based on the agents paradigm and which can seamless work and interoperate both in wired and wireless environment [2]. From the functional point of view, JADE provides the basic services necessary to distributed peer-to peer applications in the fixed and mobile environment. JADE allows each agent to dynamically discover other agents and to communicate with them according to the peer-to-peer paradigm.

The paper contains a general overview of the functionality and structure of the JADE-A-Team and a more detailed description of optimization agents including their standard functions, ontology, construction requirements and activation procedure. Further sections explain how to create and activate A-Team agents. Final section explains how the communication between agents is handled. Conclusions focus on advantages of the proposed environment and on suggestions for further research.

## 2 Overview of the JADE-A-Team

The central problem in the design of a multi-agent system is how much intelligence to place in the system and at what level. As it was observed in [3], the vast majority of the work in this field has focused on making agents more knowledgeable and able. This has been achieved by giving the deliberative agent a deeper knowledge base and ability to reason about data, giving it the ability to plan actions, negotiate with other agents, or change its strategies in response to actions of other agents. At the opposite end of the spectrum lie agent-based systems that demonstrate complex group behavior, but whose individual elements are rather simple. The JADE-A-Team belongs to the latter class.

Its main functionality is searching for the optimum solution of a given problem instance through employing a variety of the solution improvement algorithms. The search involves a sequence of the following steps:

- Generating an initial population of solutions placing them in the common memory.
- Applying solution improvement algorithms which draw individuals from the common memory and store them back after an improvement, using some user defined replacement strategy.
- Continuing reading-improving-replacing cycle until a stopping criterion is met.

To perform the above two classes of agents are used. The first class includes *OptiAgents*, which are implementations of the improvement algorithms. The second class includes *SolutionManagers*, which are agents responsible for maintenance and updating of individuals in the common memory. All agents act in parallel. Each *OptiAgent* is representing a single improvement algorithm (simulated annealing, tabu search, genetic algorithm, local search heuristics etc.). An *OptiAgent* has two basic behaviors defined. The first is sending around messages on readiness for action including the required number of individuals (solutions). The second is activated upon receiving a message from some *SolutionManager* containing the problem instance description and the required number of individuals. This behavior involves improving fitness of individuals and resending the improved ones to the sender. A *SolutionManager* is brought to life for each problem instance. Its behavior involves sending individuals to *OptiAgents* and updating the common memory.

Main assumption behind the proposed solution is its independence from a problem definition and solution algorithms. Hence, main classes *Task* and *Solution* upon which agents act, have been defined at a rather general level. Interfaces of both classes include function *ontology()*, which returns JADE's ontology designed for classes *Task* and *Solution*, respectively. Ontology in JADE is a class enabling definition of the vocabulary and semantics for the content of message exchange between agents. More precisely, an ontology defines how the class is transformed into the text message exchanged between agents and how the text message is used to construct the class (here either *Task* or *Solution)*.

The interface of the main class *Task* is composed of the following three functions: *Task()* - class constructor, *Solution createSolution()* - function generating an initial solution which can be either randomly drawn or empty, *TaskOntology ontology()* - function returning task ontology.

The interface of the main class Solution is composed of the following functions: *Solution()* - class constructor, *Solution(Task t)* - constructor producing solution to the given task, *SolutionOntology ontology()* - function returning solution ontology, *Object clone()* - function producing copy of the object, *boolean equals(Solution s)* - function returning the result of comparison between two solutions, *void evaluate()* - procedure evaluating the fitness of a solution (stored as a value of the variable *fitness*).

To obtain a solution of the particular problem instance the following actions should be carried out:

- Defining own classes *MTask* and *MSolution* inherited from *Task* and *Solution*, respectively. In these new classes constructors and other functions need to be over-ridden to assure compatibility between actions and the problem instance requirements.
- Defining own ontologies *MTaskOntology* and *MSolutionOntology* inherited from *TaskOntology* and *SolutionOntology*, respectively. Both are responsible for translating classes *MTask* and *MSolution* into text messages. Messages are more complex than a single task or solution but to produce them the outcome of *MTaskOntology* and *MSolutionOntology* class functions are used.
- Defining auxiliary classes and functions as, for example, the *Compare* function, which could be used by the *SolutionManager* to compare and sort thus far obtained solutions.

JADE-Based A-Team environment includes a number of objects shown in Fig. 1. In this paper the focus is on optimization agents.

## 3 Optimization Agents

To better illustrate the role of the above described elements, this section focuses on an *OptiAgent* actions upon receiving the *OPTIMIZE* command with respect to a list of solutions (individuals) to a particular problem instance. An *OptiAgent* is brought to life to deal with a predefined type of task through applying certain improvement algorithm to obtain the improved solutions. Such an agent (for

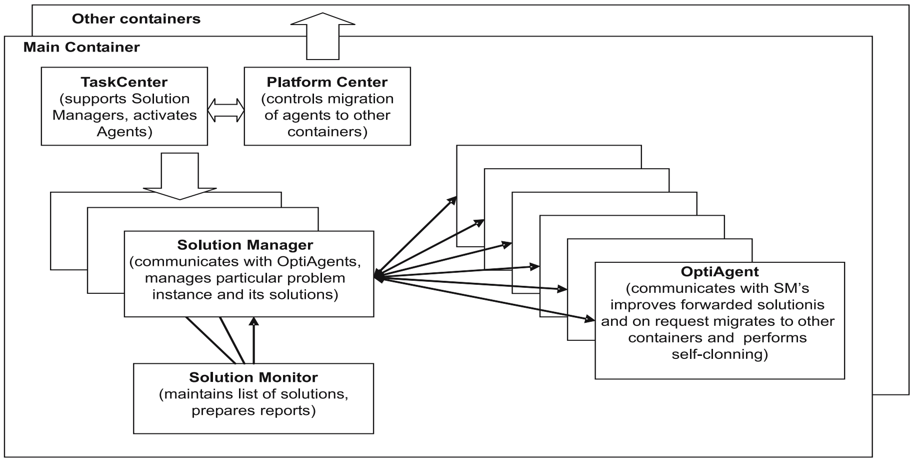
Other containers
Main Container
TaskCenter (supports Solution Managers, activates Agents)
Platform Center (controls migration of agents to other containers)
Solution Manager (communicates with OptiAgents, manages particular problem instance and its solutions)
Solution Monitor (maintains list of solutions, prepares reports)
OptiAgent (communicates with SM's improves forwarded solutionis and on request migrates to other containers and performs self-clonning)

**Fig. 1.** JADE-Based A-Team environment

example the *TSPOptiAgent* dealing with traveling salesman problem instances) over-rides three important functions from the *OptiAgent* interface:

- *taskOntology()* - a function returning class, which defines ontology of the given task type.
- *solutionOntology()* - a function returning class which defines ontology of the solution for the task with the given *TaskOntology()*.
- *optimise()* which is a particular improvement algorithm for the given task type.

Responsibility for carrying out *OPTIMIZE* command stays with the *OptimizeSolution* class, which is a private class of each *OptiAgent*. The class is an extension of the cyclic action class available within JADE. In each cycle an *OptiAgent* reads the forwarded message (if any), processes it using the *optimize()* function and sends back the results, announcing also readiness for further action.

*Example 1: Interface of the* Data *class*

```
Task t;
ArrayList ss;
// constructors
public Data() {}
public Data( Task t, ArrayList ss) {  }
// ontology functions
public Task getOpttask() {  }
public void setOpttask( Task x) {  }
public jade.util.leap.ArrayList getOptsolutions() {  }
public void setOptsolutions(jade.util.leap.ArrayList l) {  }
```

Receiving the forwarded message and resending the response requires two public classes - *Data* and *DataOntology*. Data is a public class. Its interface is shown in Ex. 1. It consists of two fields representing a task and a list of solutions, constructors and several functions required to use JADE's ontology mechanism. These are *setX* and *getX*, where *X* is defined in the *DataOntology*.

*DataOntology* defines the structure of a message and binds its parts with relevant classes. A message is a command named *OPTIMIZE*, bounded to the *Data* class and divided into *OptTask* and *OptSolutions*. The *OptTask* part is bounded to the class representing a task and the *OptSolutions* part is a list of solutions. Each solution is bounded to the class representing a single solution. The code of the discussed fragment of the *DataOntology* is shown in Ex. 2.

*Example 2: Fragment of the* Data *ontology*

```
// vocabulary
public static final String OPTIMIZE = "Optimise";
public static final String OPTIMIZE_SOLUTIONS = "OptSolutions";
public static final String OPTIMIZE_TASK = "OptTask";
// schemes
// ontology sets values of the Data class fields
add( new AgentActionScheme(OPTIMIZE), Data.class);
// add schemes of the solution ontology s and the task ontology t
s.addSchemes( (Ontology) this);
t.addSchemes( (Ontology) this);
// solutions field of the Data class is a list of solutions
// with structure defined by s
as = (ConceptScheme)getScheme( OPTIMIZE);
as.add( OPTIMISE_SOLUTIONS, (ConceptScheme)
        s.getScheme( SolutionOntology.SOLUTION),
        0, ObjectScheme.UNLIMITED);
// task field of the Data class has the structure defined by the t
as.add( OPTIMIZE_TASK, (ConceptScheme)
        t.getSchema( TaskOntology.TASK));
```

The above definition does not contain a detailed task and solution descriptions. These are taken from *t* and *s* ontology prepared for each instance of task and solution. It should be observed that when *DataOntology* is constructed both - *t* and *s* are already known. Each *OptiAgent* has been brought to life with a view to optimize a particular task, which implies defining *t* and *s* which, in turn, can be accessed using functions *TaskOntology()* and *SolutionOntology()* from the interface of the *OptiAgent* class.

Finally, the code of the *OptimizeSolutions* class is shown in Ex. 3.

*Example 3: Class* OptimizeSolutions

```
private class OptimizeSolutions extends CyclicBehaviour {
 public void action() {
  ACLMessage msg = receive();
  if (msg != null) {
  try {ContentElement ce=getContentManager().extractContent(msg);
```

```
        Concept cc = ((Action) ce).getAction();
         if( cc instanceof Data) {
          // the forwarded data are stored in the field o_data
          o_data =(Data)cc;
          optimize ();
          sendSolutions( msg.getSender()); }
          } catch( CodecException ce) { ce.printStackTrace(); }
         catch( OntologyException oe) { oe.printStackTrace(); }
        };
        ready(); //procedure of announcing agent readiness
        block( 1000); // blocking agent for a period of time
        }
}
```

## 4 Creating Agents

The proposed JADE-A-Team allows to create a variety of optimization agents searching, in parallel, for improved solutions to instances of one or more problem types and using various improvement algorithms. In Ex. 4 the operation of creating and activating an optimization agent and a solution manager is shown (in a working system there would be more agents of both types).

*Example 4: Creating and activating an* TSPOptiAgent *and a* SolutionManager

```
AgentController a=c.createNewAgent("AgentOpti1",
                "ASOP.TSP.TSPOptiAgent", null);
a.start();
Task z = new TaskTSP( "D:\\tasks\\TSP\\task1.txt");
// task will be forwarded to the SolutionManager as a parameter,
// preparation of the one-element list of parameters
Object[] _args=new Object[1];
_args[0]=(Object)z;
AgentController a=c.createNewAgent("SolutionManager",
                    "ASOP.SolutionsManager", _args);
a.start();
```

The example *TSPOptiAgent* deals with improving solutions to the traveling salesman problem (*TSP*). The *TSPOptiAgent* class is defined in such a way that it uses a particular task type (here *TaskTSP* class), a particular task solution (*SolutionTSP*) and one particular improvement algorithm for optimizing.

The *SolutionManager* is created with a view to solving an instance of this particular *TSP* problem, which is send as a parameter. However, this class does not need to "know" that the task is an instance of the *TSP*. It will simply call relevant functions defined in main classes interfaces. For example, to create an initial solution manager can call the function *Solution createSolution()* from the interface of its task parameter.

The *TSPTask* instance parameters are read from the file *task1.txt. TSPTask* class has all the required objects defined including ontology, which can be used in the *DataOntology.*

In the present version all agents are created and activated to live a single and unique life by a special agent called *TaskManager.*

## 5 Managing Communication Between Agents

Solution manager, responding to announcements knows precisely to which *OptiAgent* its message should be forwarded. Communication in the opposite direction is not that simple since optimization agents do not now which *SolutionManagers* operate on compatible tasks and solutions. To solve the problem a "yellow pages" service mechanism available in JADE has been used. It is provided by an agent called DF (*Directory Facilitator*) available in every FIPA compliant platform. Each *SolutionManager* provides the DF with its ID, the service name (*"solution management"*) and the service type. In our case the service type is a label constructed from the names of ontologies of the task and solutions that the manager operates on. The respective part of the code is shown in Ex. 5.

*Example 5: Registration of a manager in the yellow pages service*

```
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName( getAID());
ServiceDescription sd = new ServiceDescription();
sd.setType("solutions management");
// t is the task and s is a solution
sd.setName(t.ontologia().getName()+";"+s.ontologia().getName());
dfd.addServices(sd);
try { DFService.register(this, dfd);
} catch (FIPAException fe) {fe.printStackTrace();}
```

Services of the *DF* are used by optimization agents to dynamically read the list of solution managers who could be a potential addressee of messages containing announcements of readiness to act. On the list, only solution managers offering *"solution management"* service for tasks and solutions with an ontology known to the optimization agent, are placed.

## 6 Conclusions

The proposed JADE-based A-Team environment is a "middleware plus" supporting development of A-Team systems. Its advantages have been inherited from JADE. The most important advantage which is preserved in the proposed JADE-A-Team is its ability to simplify the development of the distributed A-Teams composed of autonomous entities that need to communicate and collaborate in order to achieve the working of the entire system. A software framework that hides all complexity of the distributed architecture plus a set of predefined objects are made available to users, who can focus just on the logic of the A-Team application and effectiveness of optimization algorithms rather than on middleware issues, such as discovering and contacting the entities of the system. It is expected that such an approach will result in achieving scalable, flexible, efficient, robust, adaptive and stable A-Team architectures.

During the test and verification stages JADE-A-Team has been used to implement several A-Team architectures dealing with well known combinatorial optimization problems. Functionality, ease of use and scalability of the approach have been confirmed. Further research will concentrate on providing a friendly human computer interface and developing a set of auxiliary agents that can be used to support the construction of dedicated A-Teams architectures.

**Acknowledgement.** The research was supported by the KBN, grant no. 3T11C05928

## References

1. Aydin, M.E., T.C.Fogarty (2004) Teams of autonomous agents for job-shop scheduling problems: An Experimental Study, Journal of Intelligent Manufacturing, 15(4), p. 455-462
2. Bellifemine, F., G. Caire, A. Poggi, G. Rimassa (2003) JADE. A White Paper, Exp, 3(3), p. 6-20
3. Lerman, K. (2001) Design and Mathematical Analysis of Agent-Based Systems, J.L. Rash et al. (Eds.): FAABS 2000, Springer, LNAI 1871, p. 222-234
4. Marinescu, D.C., L. Boloni (2000) A component-based architecture for problem solving environments, Mathematics and Computers in Simulation, 54, p. 279-293
5. Parunak, H.V.D. (2000) Agents in Overalls: Experiences and Issues in the Development and Deployment of Industrial Agent-Based Systems.International Journal of Cooperative Information Systems, 9(3), p. 209-228
6. Rabak, C.S., J.S. Sichman (2003) Using A-Teams to optimize automatic insertion of electronic components, Advanced Engineering Informatics 17, p. 95-106
7. Rachlin, J., R.Goodwin, S. Murthy, R. Akkiraju, F. Wu, S. Kumaran, R. Das (1999) A-Teams: An Agent Architecture for Optimization and Decision-Support, J.P. Muller et al. (Eds.): ATAL'98, LNAI 1555, Springer, p. 261-276
8. Randall, M., A. Lewis (2002) A Parallel Implementation of Ant Colony Optimization, Journal of Parallel and Distributed Computing 62, p. 1421-1432
9. Talukdar, S., L. Baerentzen, A.Gove, P. de Souza (1996) Asynchronous Teams: Co-operation Schemes for Autonomous, Computer-Based Agents, Technical Report EDRC 18-59-96, Carnegie Mellon University, Pittsburgh