Service Oriented Architecture in Multi-Agent Systems

Aaron Welch • Umut Onat • Carla Sotomayor

30 July 2008

Abstract

Large scale software programs require a lot of time and money to be devoted to them in order to ensure their reliability and efficiency, so that the duration and extent to which they can be used may be maximized. However, the scope of most large programs is rather limited, reducing them to a small handful of practical applications and requiring a new program to be built for the next unique problem. Increasing this scope to include a broader range of applications is therefore of great importance to the developers who expend time and effort to create the programs, as well as to the consumers who may otherwise spend an increased amount of time and resources with several different programs that could have been condensed into one. A Service-Oriented Architecture (SOA) presents a solution to expand upon the scope of a program, in that the program treats its functions as individual and autonomous services. What we propose to do is to apply a service-oriented approach to data collection for a generalized multi-agent system, so that multiple services may easily request to work on different sets of data independently. A proper SOA implementation could allow for a limitless number of unique services to operate on exactly what agent information end users are interested in, without any prior anticipation of particular services necessary. This will allow for users to choose what services they want, as well as develop their own for their specific needs.

1. Introduction

The main goal of a Service Oriented Architecture (SOA) is to achieve loose coupling between the two interacting agents. In our case these agents are the client and the server.

The coupling is the dependency between the server and the client. This dependency can be decomposed into two parts; real and artificial. Real dependency is the services that the client consumes from the server. It always exists and cannot be reduced. Artificial dependency is the factors that a client must comply with in order to consume the services of the server. This dependency always exists, but it or its costs can be reduced .

In our case the artificial dependency is the data transferred and connection between the client and the server. There are three main factors which might be reconsidered in order to reduce the amount of artificial dependency; the amount of data, the packaging of the data, and the connection method between the client and server.

2. Related Work

A wide variety of frameworks for multi-agent systems (MAS) with Service Oriented Architectures (SOA) have been developed in the past. Some are definitely more complex than others and provide many templates for building agents and environments. This section gives a few examples and shows there are different ways of integrating SOA into various systems. The three examples we will discuss are RePast, MASON, and Cougaar.

2.1 RePast

RePast was introduced by Social Science Research Computing at the University of Chicago. While under constant revision and extension, it has an ever growing user base. As a free, open source Java library used to create agent based simulations, RePast is extremely user friendly. There are many tools that have been added, including environments, visualization and data editing - all in two dimensions. Integration with Geographical Information Systems has also already been introduced with tutorials on how to create simple GIS models. There are many "How to Documents" on-line and programmed in support tools which help the user become more oriented with the software. Currently agents can exchange data among other agents, but group-based communication has still not been implemented.

2.2 MASON

MASON is a flexible simulation and visualization library written in Java and easily transferable across platforms because it produces identical results on each platform. Its use emphasizes on swarm multi-agent simulations with great numbers of agents. This free and open source library was created by George Mason University's Computer Science Department and the George Mason University Center for Social Complexity. MASON also has many features readily available for the user, such as its capability to generate snapshots, movies, charts and graphs of simulations.

2.3 Cougaar

Cougaar is extensively used for military purposes and it is the most complex of the three discussed in this section. It has an open source code that can virtually be used for any simulation needed. Because of its complexity, though, it isn't as user friendly as the others. It was developed for Defense Advanced Research Projects Agency (DARPA) under the Advanced Logistic Program (ALP). This completely Java based architecture is deployable on any machine and is capable of performing mass simulations of up to a hundred hosts and over a thousand agents. Agents communicate with each other to solve problems by using an asynchronous message-passing protocol that is built into Cougaar.

3. Overview

Service Oriented Architecture (SOA)is a way of constructing a system of services in order to minimize the artificial dependencies between the service provider and the service consumer. The

major areas that SOA applies are business processes and web services. Cognitive Agents for Social Environments, or CASE, is a multi agent system developed by the computer science department of Trinity University. Inside CASE, the local services are going to be oriented in order to increase the system efficiency. A ServiceManager class defines the connection protocol among the clients and the Multi-Agent System. There is also a Service interface to define the behaviors of different services socket services and file services. The basic connection between the clients and the MasterServer through the ServiceManager can be seen at Figure 3.1.

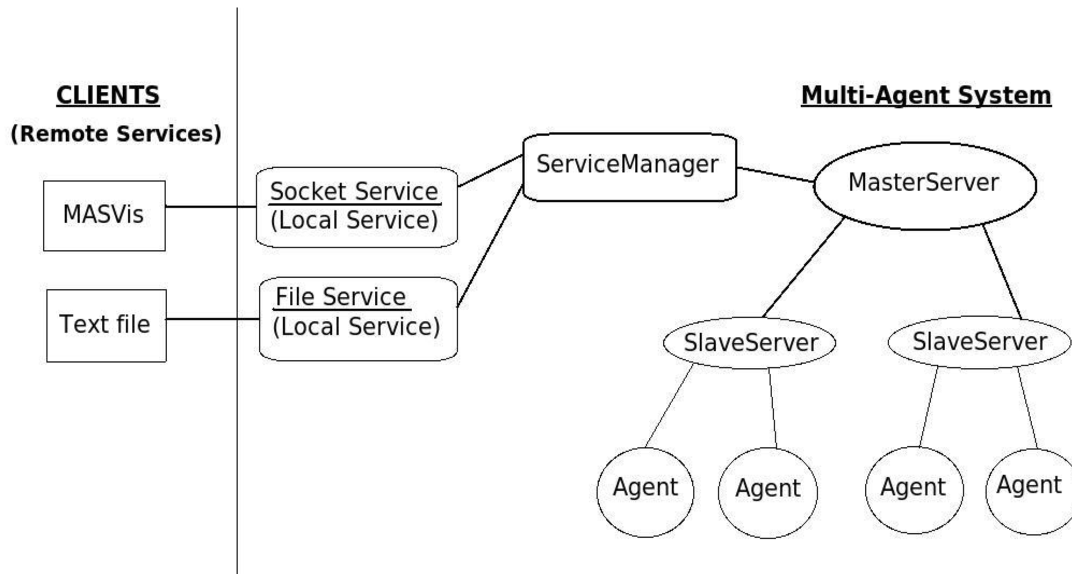

Figure 3.1

In order to build an efficient SOA, we need to overcome some issues. There are four main problems that require our attention at this point:

3.1 Local Services

To establish a connection between remote services and CASE, the use of local services as intermediaries is necessary for creating different kinds of connections and facilitating communication. Of a number of possible local service types, the most universally usable would be a socket service that allows users to connect to specific ports. Communication across sockets is fast and widely used. In addition to a socket service, CASE provides a file service to record the simulation data to disk for later viewing.

3.2 Data Packaging

There are a few possible options for packaging the requested information and sending it to the clients. Factors of them that need to be considered in choosing the best one for any given situation include processing and communication speed, interoperability with remote service formats, and restrictions on usability. The most efficient way to send the packages over the network is using binary format for the data. It provides the fastest I/O procedure. CASE sends all of its output in binary format. However, not all of the clients might accept the data in binary format, like Geographic Information Systems (G.I.S.), and may require too much work from the

users and remote service programmers. What may be the most widely usable method of packaging is the use of XML to transmit data. Unfortunately, this would also be the slowest way of sending information, partly due to the large size of the files being sent across the network, and the use of strings of text to store everything. This general format could be easily applied to a wide variety of remote services with little effort required from end users. Another option is a plain text format, without the tags and formatting of XML. This would reduce some of the interoperability of the XML, but also greatly reduce file size, making it the ideal packaging method for tracking and logging interactions.

3.3 Selecting Agents

When a client requests information on certain agents we face another problem. The ServiceManager and the servers do not have any knowledge regarding the agents or their status. In order to overcome this, we need a method to select agents based on the information they may or may not have, while suffering minimal performance loss. Therefore, it is necessary to be able to discover the contents of an agent without initially knowing its type. Furthermore, this check needs to be executed every time that information is requested, in the case that agents change or new agents appear. An issue that arises with this is that the remote service must specify exactly what information it is interested in. To clearly detail what agents a remote service wants, boolean statements can be evaluated, comparing values to variables for the sake of isolating elements of particular states. In addition to this capability of specifying what information a remote service is interested in, it needs to be able to choose when to get the information. As such, remote services must be able to specify an interval, so that information is gathered only when the current time step matches with the interval, such as every fifth step.

3.4 Interface

Once this implementation is completed, CASE must present a proper interface to allow users to access and change service data. To do this, some options need to be viewable and editable from the CASE GUI that will allow for adding and removing of remote services and their respective class paths, and for customization of output file options. In addition, in order to more fully incorporate Java-based remote services, the remote services must implement various GUI related methods . These methods must be outlined so that the expectations from any remote service are the same. With this functionality included, many key features can be controlled directly from CASE, allowing for a more centralized control scheme. In considering such an interface, user friendliness is of utmost importance in order to ensure maximal usability. Since a lot of this functionality is implemented in a general and abstract way, friendliness of the system may be a difficult issue to handle.

3.5 Relocating the Master Server

The master server was built on a remote machine. This was causing a problem for the communication between the master server and the slave servers. It also caused problems when a service would try to change information being sent.

4. Solutions

4.1 Local Services

CASE provides two local services; a socket service and a file service. Both the SocketService class and the FileService class implement the Service interface. The SocketService class provides a socket connection between the Multi-Agent System and the remote services. The FileService class outputs the data in a file in a simple text format. (Figure 4.1.1).

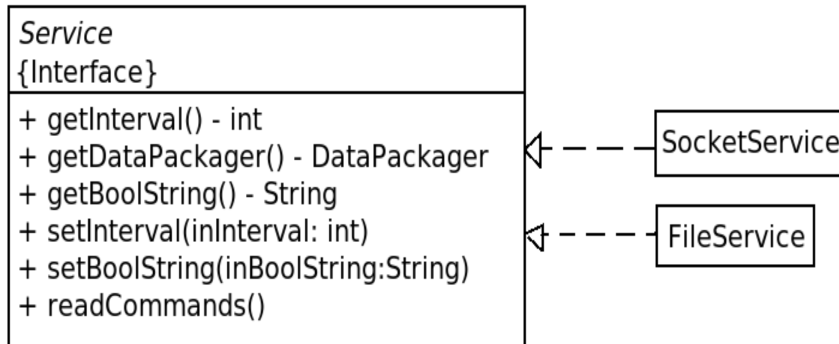

Figure 4.1.1

The main purpose of the SocketService is to create a socket connection between the remote service and the multi-agent system. It also defines the packaging type depending on the first value that the remote service sends in. The default packaging type is binary. The default interval value is set to five and the default value for the boolean statement is set to true. The SocketService class provides set methods to change these values. The readCommands method will be used to call the readCommands method of its DataPackager (Figure4.1.2).

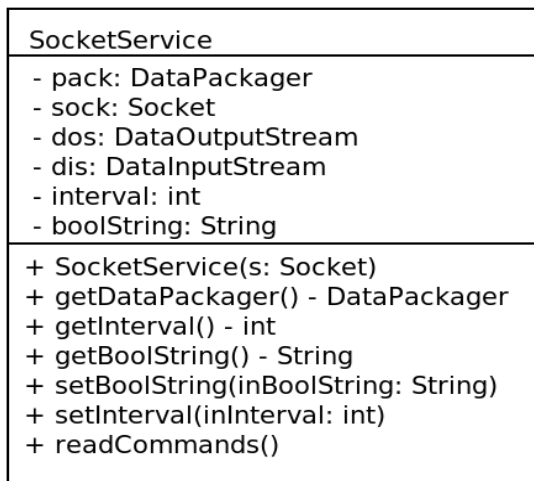

Figure4.1.2

The FileService's main purpose is to store the output in a file in simple text format. The FileService class takes in an integer value to define the type of packaging. The default packaging type is simple text format. However, it can also store the output in XML format, given a fully

functional XML packager. The boolean statement, the packaging type, and the interval values can be changed through the interface. It also takes in a file object as a parameter to write the output. The readCommands method does not have any functionality under the FileService class; thus has an empty body(Figure 4.1.3).

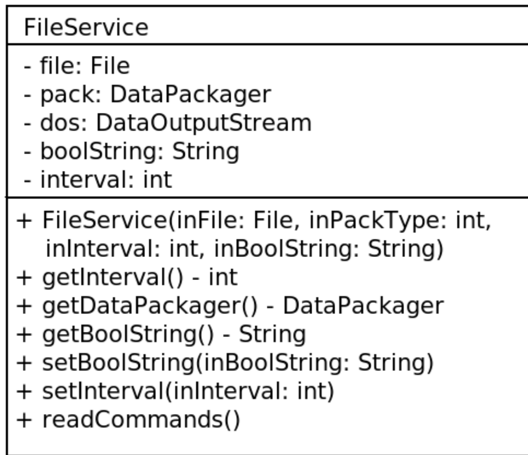

Figure 4.1.3

4.2 Data Packaging

For the data packaging problem, CASE provides three packaging options; a binary packager, an XML packager, and a text packager. All of these classes implement the DataPackager class(Figure4.2.1).

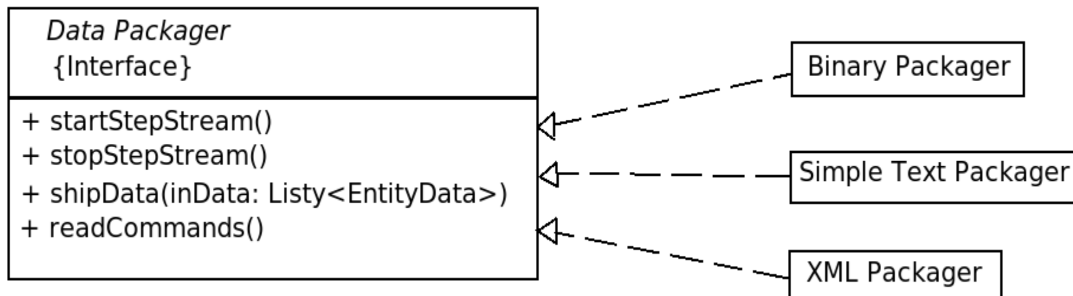

Figure4.2.1

The shipData method sends the data over the output stream and the readCommands method allows the remote services, like SwiftVis, to send in information about the interval and the boolean statements. The startStepStream and stopStepStream methods indicates the beginning and the end of the data transfer for each step. The interval value defines the frequency of the data transfer; the amount of data to be send between each step. The boolean statement is used in agent selection process. Therefore, while the readCommands method defines the characteristics of the

entities to be selected and how often to send the data, the shipData method sends the requested information to the remote services.

Furthermore, CASE has an EntityData class that holds the information regarding the entities. This class collects the ID and location values of the entity as a default feature. The EntityData class is used inside the packagers in order to collect and ship the data(Figure 4.2.2).

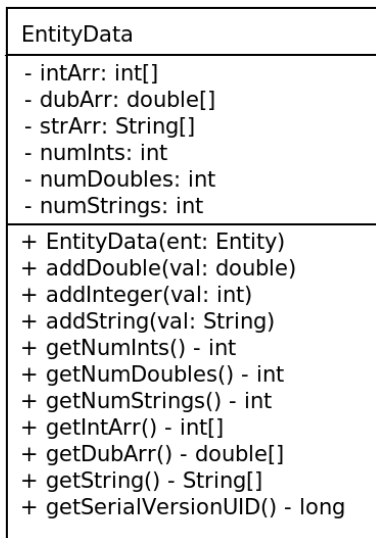

Figure 4.2.2

The BinaryPackager class writes to an output stream in binary format. The shipData method gets a list of EntityData and ships the incoming information through the data output stream. First, it sends the integer values for the entities, and then the double values, and finally the string values regarding the entities. The readCommands method listens to the input stream. If the remote service would like to set the interval value, then it must send in zero, and the next incoming integer value will set the interval. If the readCommands method receives one through its input stream, then the next incoming value will be the boolean statement. The next incoming integer value will set the length of a byte array; which is the incoming message in byte format. The byte array will construct a string which will be the boolean statement to be used in agent selection process(Figure 4.2.3).

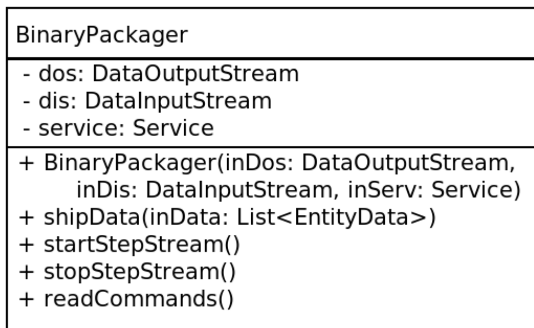

Figure 4.2.3

The XMLPackager class sends the data in XMLformat. The shipData method ships the

information through an output stream writer wrapped around the data output stream. First, it will send the integer values for the entities and then the double values, and finally the string values; all of which will be in XML format. The readCommands method will apply the same logic as the BinaryPackager. However, it expects the incoming values to be in XML format.
This part of the code is under development, and is listed under the future works section of this report.

The SimpleTextPackager class helps the FileService to create an output file. The shipData method of the SimpleTextPackager prints the data to a print stream wrapped around the data output stream. It doesn't take in any commands, thus its readCommands method has an empty body(Figure 4.2.4).

```
SimpleTextPackager
-------------------------------------------------------
- ps: PrintStream
-------------------------------------------------------
+ SimpleTextPackager(inDos: DataOutputStream)
+ shipData(inData: List<EntityData>)
+ startStepStream()
+ stopStepStream()
+ readCommands()
```

Figure 4.2.4

4.3 Selecting Agents

In order to overcome the agent selection problem, CASE includes a BooleanParser and an ArithmeticFormulaParser to parse the string input of the users, allowing users to enter boolean expressions. The users are now able to enter a boolean expression regarding the agents and will get the output which holds true for the statement.

The BooleanParser and the ArithmeticFormulaParser classes are modified versions of the BooleanFormula and the DataFormula classes inside the SwiftVis, a visualization tool developed by Dr. Mark Lewis. The difference between CASE parsers and the ones in the SwiftVis is that CASE parsers are able to communicate with the entities by using reflection. The BooleanParser class is able to evaluate boolean expressions in the string form where as the ArithmeticFormulaParser is used to evaluate arithmetic expressions in the string form.

For example, to collect the entities within a certain region, say x coordinate is between zero and five and y coordinate is between zero and 8, the user can enter a boolean string: “getLocation.getX < 2+3 && getLocation.getY < 4*2”. The parsers are able to parse this input and evaluate it. They have reflective methods inside that are able call getLocation.getX and getLocation.getY methods on agents, and evaluate the whole string. As a result only the information regarding the agents inside the specified area is collected.

Both of CASE parsers are using reflection. Therefore, users can collect the information not only from the methods that return a numerical value but also the methods which return a boolean value. For example, given a foreclosure simulation, if the users want to collect the data regarding

only the agents that are foreclosed, the only thing that they need to do is to send the name of the method. For this case let's assume it is “isForclosed”. This time only the data regarding the agents whose isForclosed method returns true, will be send to the remote service. The requirements for boolean statements are that the users must know the methods that the entities have, and the remote services must be able to send boolean statements.

4.4 Interface

To make the interface more user friendly and not so hard coded, CASE provides an Options window. Under the File tab, the system has a working “Options” option right above the “Exit” option. Selecting the “Options” directs users to “Multi-Agent System Options” interface where they can edit simulations, servers, services, and file output.

Under the “Simulations” tab the users can add simulations or remove them. Previously, the simulations were hard coded inside a text file. Hard coding is removed, and now the simulations that are going to be used can be edited by the users.

The code for the SwiftVis, which was a remote service, is completely removed out of CASE. Now, if the users would like to add remote services, they can use the “Services” tab under the “Options”. This tab allows users to add or remove remote services, like SwiftVis. The “Class Path” tab allows users to add or remove class paths that are associated with the remote services.

The users are also able to add more servers or remove a current machine out of the servers list by using the “Servers” tab. This tab updates the servers list shown under the “Server Controls” frame.

The final tab is the “Output File” tab under the “Options”. This tab allows users to edit filing options for the simulation. The users can select to store the output in a text file. They can also define the interval and the boolean statement for their output. By using the “Add Output File” option, the users can also select a file to write the output.

4.5 Relocating the Master Server

The master server had to be relocates on the local machine. This was done in order to optimize the communication between the master server and the slave servers. The simulation and communication to and from a service could be done on the local machine greatly improving efficiency.

5. Experiments

As various implementations of packagers, services, and ways to connect the two emerged. Ways to test them had to emerge as well. Simple testing programs were created for that sole purpose and usually made some bugs apparent. Since MAS has a Service Oriented Architecture, a test service program was needed to test the implementation of packaging data and sending it over a socket. Therefore, SwiftVis was used as a data visualization tool. The following sections cover

the experimentations done to test our SOA.

5.1 Sockets and Packagers

When trying to decide how to collect the data from the entities and where to format it, some issues appeared with the Master and Overlord Servers. The code had to be changed to make the Master Server and Client instantiate on the local host machine. Once the Master Server was placed on the local host machine, testing was made easier. To make formatting easily selectable by various services, the Slave Servers collect all of the Entities data in an EntityData object and, in order, the Slaves send the List of EntityData objects to the Master Server which then sends formatted data to the services. To begin testing early on, agile methods were used. A Simple Service was created to test the socket connection and the Binary Data Packager to verify the data from the agents was being correctly packaged and sent across the socket from all slave servers. The socket port number was hard coded in as 4444 and some hard coding was also involved to always package the data in binary format; therefore labels were also needed for the various numbers coming through.

5.2 Correcting Simple Simulation

Once the sockets and data packagers were working correctly, SwiftVis was used to test the Simple Simulation being used – The Game of Life. When the plot was brought up during the simulation, it was soon apparent that the agents weren't being placed in a manner suitable for The Game of Life. It seemed that prior testing for Load Balancing left one Slave Server with the task of creating hundreds more agents than the other Slave Servers. The agents also weren't being placed in a grid like manner; they were randomly being placed throughout the grid. Another problem made apparent by using a Sink in SwiftVis was the creation of 30 or so more agents after the first step in the simulation. These issues were tested and fixed because of the sockets and data packagers being used with a service to collect the necessary information from the agents and verify the simulation was running as it should be.

5.3 Options Interface

Once the Options interface was created, it was fairly easy to test. The Options window is used to set several settings, such as servers, services, simulations, and class paths. The text files, which were manually edited to contain this information, were deleted and the Interface Control was edited to not pull from those files. All the settings were then changed in the Options window and saved to be brought up when the MAS interface was reopened. SwiftVis, once completely integrated into the MAS project, was taken out. It was added as a Service in the Options interface and the class path was connected.

5.4 Agent Selection with Boolean Statements

The File Service was created at the same time as boolean statements. In the Options interface, an Output File tab was created to ask the user if a data output file was necessary and, if so, the user could input the file in which to output the information, the packager in which the data should be collected, the interval between time steps to collect, and a boolean statement for which agents to

collect data from, if any. The Simple Text Packager was created solely for the purpose of the Output file and user readability. To test the boolean statements, an Output text file was created with the Simple Text Packager, an interval of 10, and a boolean statement of "getPosition.getX = 5" so as to just collect the data from the entities located at positions with an X value of 5. The same test was later done remotely with SwiftVis. Both tests encountered no errors.

5.5 Remote Services

After the hard coding of host names were taken out, testing the connection of a service from a remote machine was made possible. Simple Service was able to run on one computer while the simulation was running on another. Further testing was made possible by input boxes added to SwiftVis to take in a host name, interval, and boolean expression. No issues occurred when connecting all of the test Services created along the way as well as SwiftVis. They were able to connect at the same time as well without any problem.

6. Future Work

6.1 XML Packager

The current version of the XMLPackager class is unable to read incoming commands. It is also missing a parser. Simple API for XML, or SAX, is considered as an efficient option for handling XML documents. Xerces is chosen to be the parser. However, the size caused some issues. Other parsers, like Piccolo, might be considered. Piccolo is a very small parser and only works with SAX.

To be able to have a working SAX, CASE also needs a content handler class. This class should help with handling the contents of incoming XML commands. The handler class must be able to identify the contents of the incoming XML document and should be able to identify the end of the document. The design should also allow remote services to send multiple XML documents for different commands without breaking the connection to CASE.

6.2 Load Balancing

Load Balancing adjusts server boundaries as needed to keep all servers ending time steps together. It would be inefficient for one server's load to take twice as long as the others and not have the code to handle the changes. For this reason, a LoadBalancer class was added to CASE.

The LoadBalancer currently implemented is error prone and does not keep all the agents it should. Agents are lost when the server boundaries change and the reason is still unknown. Future students working on this project could debug the current implementation or create their own that is more efficient, but currently the servers do not load balance so as to preserve the amount of agents in a simulation.

References

- Feng Lu and Kris Bubendorfer, *“A RMI Protocol for Aglets”*. In Vladimir Estivill-Castro, editor, *Twenty-Seventh Australasian Computer Science Conference (ACSC2004)*, volume 26 of *CRPIT*, pages 249-253, Dunedin, New Zealand, 2004. ACS.

- Elena Gómez-Martín, Sergio Ilarri and Jose Merseguer, "Performance Analysis of Mobile Agent Tracking Approaches", Sixth International Workshop on Software and Performance (WOSP'07), Buenos Aires (Argentina), ACM Press, pp. 181-188, 2007.