Interoperability in Multi-Agent Systems: Lessons Learned

Marco Antonio Furlan de Souza[1], Jomi Fred Hübner[2] *,
Jaime Simão Sichman[2] **, and Maria Alice Grigas Varella Ferreira[1]

[1] Laboratório de Tecnologias de Software
{marco.souza, maria.alice.ferreira}@poli.usp.br
[2] Laboratório de Técnicas Inteligentes
{jomi.hubner, jaime.sichman}@poli.usp.br
Escola Politécnica da Universidade de São Paulo
Av. Prof Luciano Gualberto, 158, tv.3
05508-900 São Paulo, SP

Abstract. Today there is a substantial number of Multi-Agent Systems (MAS) tools available to the agent developer. Part of them, for reasons of performance and development easiness, were built according to some particular language/protocol, making difficult or hindering the communication of its agents with others developed with different languages and protocols. Despite the increasing effort in the definition of standards for agent communication protocols and languages, the heterogeneity in MAS is a fact that is there to stay. This paper reports experiences of turning an MAS tool, Saci, interoperable. The solution is based on a CORBA bridge that enabled agents written in CORBA-mapped languages to communicate with other native Saci agents, without changing its architecture or programming style.
Keywords. Multi-Agent Systems Interoperability, Multi-Agent Systems Tools and Programming, Multi-Agent Systems Architecture.

1 Introduction

Interoperability is still a great problem in Multi-Agent Systems (MAS) tools. Besides the existence of standardization efforts as practiced by FIPA [3, 4, 5], most of MAS tools available are not interoperable. This lack of interoperability in MAS tools are in part due to design decisions and to the tools employed in their construction. The former implies in an MAS tool with its own specific architecture features like agent registration, advertisement capabilities, white and yellow page services, agent communication language, to mention some, while the last forces the agent designer to use the same implementation language and protocol used in the original tool. Although creating agents in that manner has

* Supported by Universidade Regional de Blumenau and CAPES
** Partially supported by CNPq, grant number 301041/95-4 and by CNPq/NSF PROTEM-CC MAPPEL project, grant number 680033/99-8

benefits like obtaining good performance and easy use, it restricts the development of agents to a single language and/or protocol, thus restricting the use of the MAS tool.

This paper reports experiences with MAS interoperability in the design of a CORBA interface to the Saci system [6, 7]. Saci (Simple Agent Communication Infrastructure) is a tool that facilitates the task of programming communication among distributed agents, providing an API to manipulate KQML messages and tools useful in a distributed environment. Saci was written in Java with RMI (Remote Method Invocation) as distributed communication protocol and this fact forces the developer to use the same tools in his/her agents. To overcome this problem, a CORBA bridge was implemented to the Saci tool. CORBA is the acronym for Common Object Request Broker Architecture, an Object Management Group (OMG), an open and vendor-independent architecture and infrastructure enabling computer applications to work together over networks [9]. It uses a standard protocol named Internet Inter-ORB Protocol (IIOP). So applications can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network.

The use of a CORBA bridge to Saci tool preserved Saci's original architecture (easy to understand and to use) and maintained its style of writing agents, enabling developers to write Saci agents in other languages (CORBA-mapped) in a similar way as with Java/RMI.

This paper is organized as follows. In section 2 a brief view of interoperability with CORBA is given, just to position concepts related to this technology; in section 3, experiences are related with the development of the Saci's CORBA bridge, describing the original tool, the design decisions that guided the implementation, the CORBA interfaces, the implementation of the bridge and how to write CORBA-Saci agents; finally, in section 4, a discussion on the pros and cons of the implementation concludes the work.

2 Interoperability with CORBA

The first step toward the construction of any distributed system in CORBA requires the definition of an interface of methods which will be executed by the target remote objects [13, 11, 10]. In this phase, a text file describing the interfaces in the *OMG Interface Definition Language* (OMG IDL) must be written. The OMG IDL is an independent declarative programming language, similar in structure to C++, that supports the declarations of interfaces, methods, structured and basic types, as well as object-oriented concepts like inheritance, single or multiple [13].

After the definition of the interfaces and with the help of an IDL compiler, the IDL file is translated into the desired target language. It depends, of course, on the support of the CORBA-mapped languages by the compiler. The IDL compiler generates source code for the two sides: the *client side* (also denominated "stub") and the *server side* (also denominated "skeleton"). The "skeleton" is

used as a basis for the server implementation – where the logic of the objects is realized – and the "stub" contains code to facilitate the creation of client applications. These source codes encapsulates all the necessary machinery to establish communication among clients of remote objects and its servers with the IIOP protocol. To render this, each side must connect its objects on the ORB (Object Request Broker) which handles the remote calls. Figure 1 summarizes the functioning of a typical CORBA system [9].

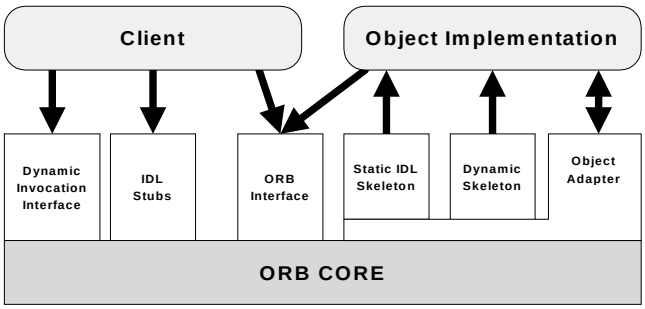

Fig. 1. The structure of object request interfaces

Figure 1 shows a client application that must be connected to its ORB in order to communicate with an object server. Requests to remote objects are made using its "stubs" (generated according to the target objects) and/or its *dynamic invocation interface* (providing independence from the interface of the target object). On the server side, besides the necessary existence of the ORB, the *static IDL skeleton* handles the "up-calls" of objects methods, that is, their execution. The *dynamic skeleton* does the same job as the static IDL skeleton, but for dynamic method invocations from the clients. Finally, the *object adapter* defines how an object is activated, that is, the process or thread architecture that will be used to manipulate the objects on the server side. OMG has defined a *basic object adapter* (BOA) and the *portable object adapter*(POA), to avoid the proliferation of different adapters [9].

3 Case Study: The CORBA Interface to Saci

In this section the experiences with the implementation of a CORBA bridge to the Saci system are discussed. First, a brief view of the Saci system is given, explaining its architectural principles and usage. In sections 3.2, 3.3 and 3.5 the focus is on the design of the CORBA bridge to Saci, explaining the design decisions that guided the project, the CORBA interfaces to Saci and finally how to write CORBA-Saci agents with this architecture.

3.1 Saci System

Saci (Simple Agent Communication Infrastructure) is a tool that facilitates the task of programming communication among distributed agents [6, 7]. The main features provided by Saci are an API to manipulate KQML messages [8] like composing, sending and receiving and tools to provide useful services in a distributed environment like agent name service, directory service, remote launching, communication debug, to name a few.

The internal behavior of Saci agents is simple, since it is based on an intuitive society model. The agents have to enter a society; then, exchange messages and announce skills; and at last, exit the society. Each society has one, and only, facilitator – an agent with special features – in order to help agents to meet each other. The facilitator has a list of agent identifications (and their respective network address as a Java/RMI stub) and a list of skills available in the society (and the respective set of agents that are able to perform those skills). Thus, to enter (or exit) a society, an agent should ask permission to the society facilitator. Once inside the society, Saci agents may exchange messages with each other and/or announce skills to the facilitator.

To exchange messages, Saci agents have an entity called MBox (standing for Mail Box) where the messages received for processing are queued. The MBox interface provides several methods to select messages that arrives to the agent (e.g., gets the first message in the MBox; gets the first message that matches a pattern; waits for a message; wait for n messages that matches a pattern). When an agent (A) asks its MBox to send a message (M) to another agent (B), A's MBox asks the facilitator for B's localization. Once received, this localization (the network address from the agent name service), the agent pair (A and B) can communicate directly exchanging KQML messages (see Figure 2).

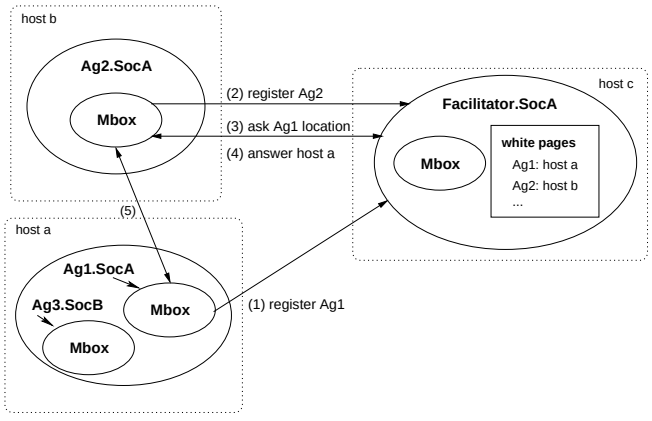

Fig. 2. Agent name service

The announcement of agent skills obeys the same process of entering the society, but here the already set agent identification will be stored together with its specific skills on a list called yellow-pages that will be used to provide the directory service. When an agent needs someone to do a specific job, it asks the facilitator for somebody who is able to perform it. After querying the yellow-page list, the facilitator sends a list of agents back to the asker agent. By now on, the agent will ask the services directly to the provider agent (see Figure 3).

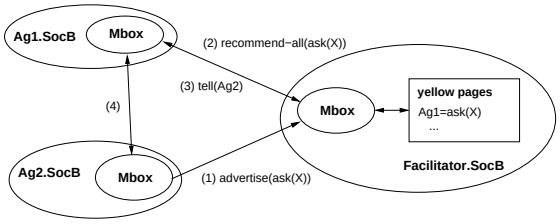

Fig. 3. Directory service

3.2 Design Decisions

The main design principle adopted in the construction of the CORBA interface to Saci was the preservation of the Saci's original architecture and interface. This decision was taken due for two reasons: to preserve the Saci style of agent programming and to keep using its services. Preserving the Saci style of programming help Saci programmers to easily migrate to CORBA versions while keeping using its services (agent name service, directory service, remote launching, communication debug and others) brings to CORBA agents a set of working services, without the necessity to build them from scratch [6, 7].

A more convenient way to realize these ideas is by the way of a CORBA *bridge* architecture, inspired by the Bridge Pattern from Gamma et al. [2]. Basically, the bridge is implemented in a CORBA server that accepts connections and invocations from CORBA clients and translates these invocations to real Saci's agents. For each CORBA agent in the server there is an associated Saci agent that really do the job, so this agent serves like a *proxy* to CORBA agents' requests [2, 12]. CORBA clients of this bridge can behave as Saci's agent server or clients.

In Figure 4 two Saci agents are created and then manipulated indirectly. They are delegates of CORBA server agents which in its turn were created by CORBA agent client programs (consider Java and C++). To clients, the behavior is the same as they were accessing Saci agents directly. The special server object named AgentManager is used in this architecture just to maintain a pool of agents and to enable the client programs to request a reference to a CORBA agent [11]. This is justified by the fact that each remote agent will receive connections in IIOP

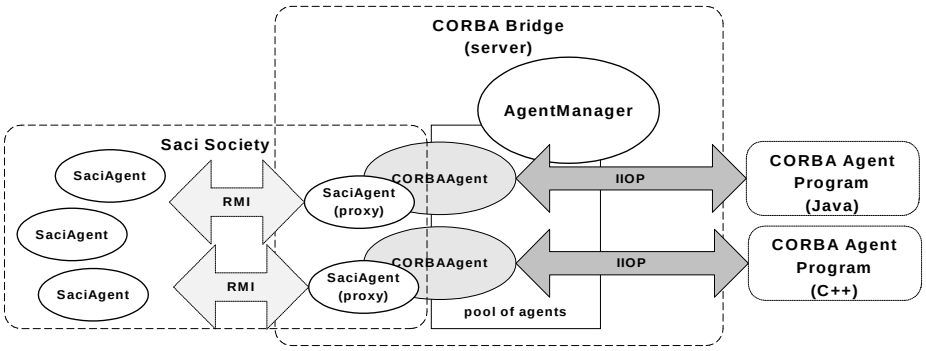

Fig. 4. The bridge architecture to Saci

protocol and dispatch the requests in RMI protocol, so there is a necessity to put each agent in its own thread of execution, establishing a separate communication channel for each remote agent. The bridge is a CORBA server written in Java, so this enables the direct use of a Saci agent (the proxy) like an ordinary Saci agent in the system.

3.3 Saci-CORBA Interfaces

Following the design principles described in section 3.2, a CORBA IDL file was written to support the interfaces to Saci system. Two interfaces were defined: CAgent and CAgentManager. The interface CAgent encapsulates all the functions of a Saci agent [6] while the interface CAgentManager has just two functions: `reserveAgent`, responsible for supplying agent references to clients and `releaseAgent`, used by clients to free the agent reference after its use.

Additionally, due to type mismatch between CORBA types and Saci types (which are Java types) two types were defined in the IDL file: CMessage and CVector. The type CMessage is just an alias to the CORBA string type. It is used to store KQML messages. In Saci, there is a type Message to store messages, which is based on a Java Hashtable type. Since in CORBA this type does not exist, all the KQML messages are manipulated by CORBA agents in a plain string format. The type CVector represents a vector of strings, used to store results from Saci functions like `consultYP` which returns a list of candidate agents to a service.

3.4 The CORBA Bridge to Saci

To implement the CORBA bridge, it was necessary to use an IDL compiler as well an ORB implementation. The choice was to use the EngineRoom ORB (http://www.engroom.com) because its ORB and IDL compiler are free for non-comercial use and have CORBA language mappings to C, C++, Java and Perl.

The bridge was implemented in Java and for the server side, the IDL compiler has generated, among other files, the Java classes CAgent and CAgentManager. These are just interfaces in Java, a direct translation from the IDL file, so they must be implemented. To implement these classes, the two skeleton classes generated by the IDL compiler, _CAgentImplBase and_CAgentManagerImplBase, were extended with appropriate code, implementing the logic of the objects.

The class CAgentImpl was written to implement the methods of the CAgent class. This class was inherited from _CAgentManagerImplBase class and internally has an object of type Agent, imported from the Saci package. This is the proxy agent that will enter and communicate with other Saci agents in a society. These relationships are shown in Figure 5 using an UML class diagram [1].

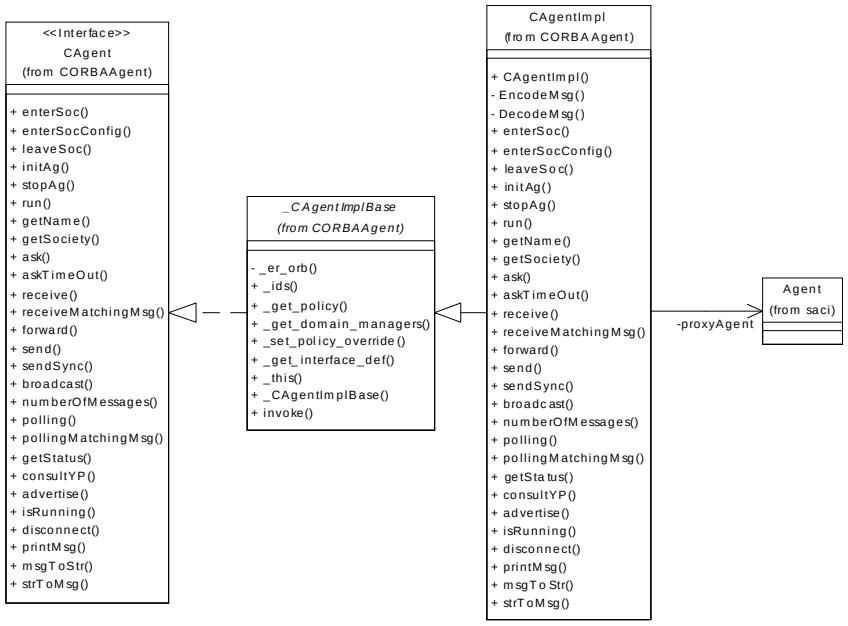

Fig. 5. Relationships of the CORBA CAgent classes

Note that, in Figure 5, each method of the CAgent class interface is implemented in CAgentImpl class (the methods signatures were omitted for simplicity). The implementation of each method is very simple because it delegates its execution to the proxyAgent object. For example, to send messages to other agents, a CORBA agent can use the method `send` which, in its turn, will execute the method `getMBox` from a Saci agent object, indirectly retrieving a reference to its mail box interface. With this reference it is possible to use the method `sendMsg`, to send a new message to another Saci agent. This collaboration is

illustrated in Figure 6. All other methods of a Saci agent are manipulated this way.

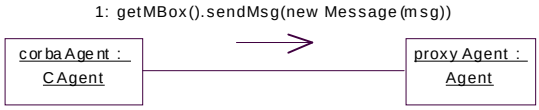

Fig. 6. The use of the proxy agent to send messages

The type mismatch between the CORBA type of message (CMessage) and the expected type of `sendMsg` (Message) was solved by one of the constructors of the class Message from Saci, which converts a plain string into a Message type. In a reverse process, for example when using the `receive` method, its is necessary to convert a Saci message to a plain string (CMessage). Fortunately, the Message class in Saci has the method `toString` which converts its internal format into a plain string, as indicated in Figure 7.

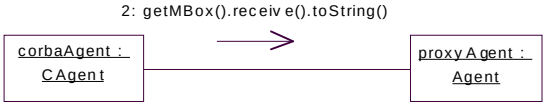

Fig. 7. The use of the proxy agent to receive messages

It is a task to client programs to parse KQML messages from the result of `receive`. CORBA agent programs in Java can use the KQML parser class from Saci to accomplish this. CORBA agent programs in other languages (like C++) must use its own KQML parser.

Now, considering the implementation of CAgentManagerImpl class, it was decided to use a Java vector as a pool of agents. The idea is to start the pool with an initial number of agents (which incrementally increases). Client applications can get agent references by the use of `reserveAgent` method, which connects the reserved agents to the ORB. After using the objects, client applications should release these objects to the ORB executing the Agent Manager's `releaseAgent` method. The relationships of the implementation classes of the CAgentManager class are illustrated in Figure 8.

Finally, the CORBA bridge is implemented as a Java "daemon" which creates and connects a CAgentManager object and waits for clients. It is implemented as a Java class named CORBAServer which requires just one command-line parameter that indicates the number of CAgent objects that must be initially in the agent pool (if omitted, the number is five). In its implementation, the CORBA bridge depends on a name server to locate and identify the objects

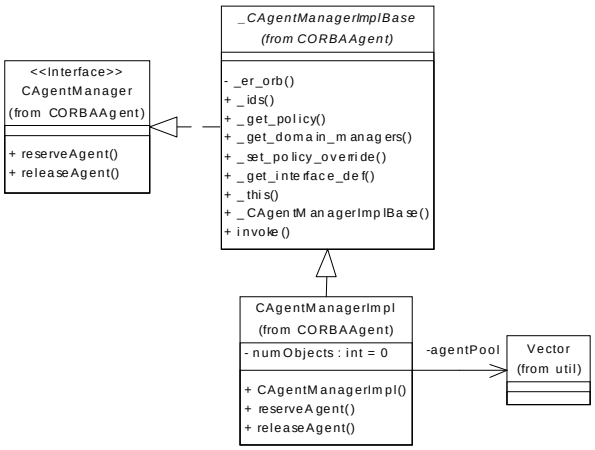

Fig. 8. Relationships of the CORBA CAgentManager classes

properly. In the tests, the `tnameserv` program from Sun's Java 1.2 SDK were used as a name server.

Basically, when this deamon is started, the following operations are executed:

1. ORB initialization;
2. CAgentManager object creation;
3. Connection of the CAgentManager object to the ORB;
4. Enter in a listening state - it waits for client invocations.

At step two, when the CAgentManager object is constructed, it creates a pool of CAgent objects but does not connect them to the ORB. This connection just occurs when client applications use its `reserveAgent` method. When a CAgent object is reserved by some client application, it is ready to be used as a typical Saci agent.

3.5 Writing Saci-CORBA Agents

Saci-CORBA agents are client applications that use the CORBA server to communicate with other Saci agents. These applications must use the "stubs" generated by the IDL file to get references to the CAgentManager object and CAgent object. Basically, a CORBA agent program must execute these operations before start using the CORBA agent:

1. Initialize its ORB;
2. Use the CORBA Name Service to get a CAgentManager object reference;
3. Invoke the `reserveAgent` methot of the CAgentManager object to get a reference to CAgent object;
4. Use the CAgent object, implementing the agent program.

5. Release the CAgent object (optional).

Since steps one to three are common to all CORBA agent programs, a class named CORBABasicAgent was written to facilitate the creation of CORBA agent programs, available at moment in Java and C++. This class has aggregated two references for objects CAgent and CAgentManager and its constructor do all the ORB initialization and resolution of its aggregated object references. All the methods available in CAgent interface are also available in the class CORBABasicAgent, so the task to CORBA-Saci agent designers is just to inherit from this class and implement the method `run` with the appropriate agent code and create a program to call it [6]. The relationships among these classes are shown in Figure 9.

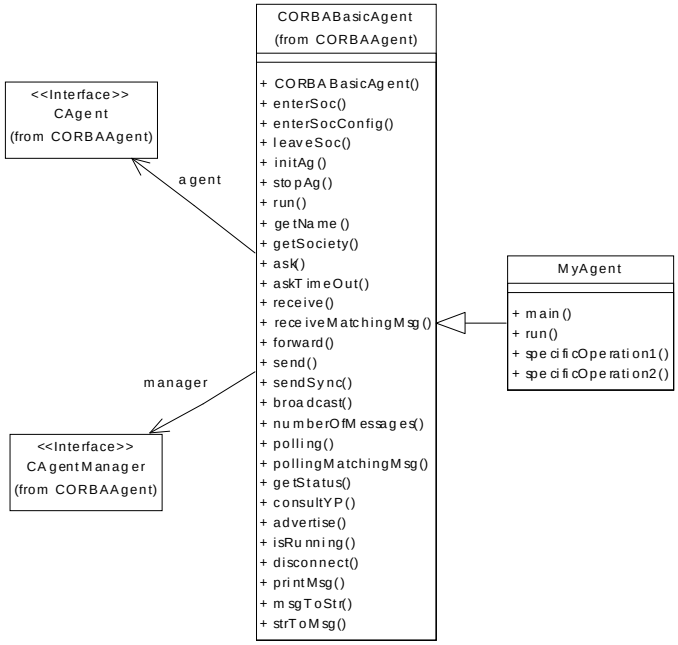

Fig. 9. The CORBABasicAgent class

In Figure 9 a class named MyAgent, inherited from class CORBABasicAgent, represents a Java class that executes some agent program. The method `run` implements the logic of the agent while the `specifcOperation1` and `specifcOperation2` represent specific operations that this agent must execute. The execution is properly done inside its `main` function. Agents in other programming languages can be implemented in a similar way.

4 Conclusions

This work presented a non-intrusive technique based on a CORBA bridge to make an MAS tool interoperable. The first lesson learned from the work is that, although it is not an unique solution to MAS interoperability, our solution has benefits like preserving the original architecture and programming. In fact, if changes could be made to Saci architecture to enclose the CORBA interface, it would imply in a complete rewriting of large parts of the system, forcing the applications already existent to be rewritten.

Second, as a consequence of the use of CORBA, it is possible to create Saci agents in languages other than the original, depending, of course, on the availability of CORBA-mapped languages and its ORB implementations. Since this CORBA interfaces preserved the "look-and-feel" of the original system, it is possible for developers in other programming languages to write Saci agents following the same instructions present in the Saci programming manual [6].

Third, just small scale tests with CORBA agents and CORBA bridge were conducted and the results were very satisfactory. No substantial performance degradation of CORBA agents in relation to original Saci agents has been noticed, and this results directly from the simplicity of the bridge implementation.

Now, some drawbacks of this implementation. First, it is a task to the developer to parse KQML messages from his/her Saci-CORBA agents. If using Java, this is easily solved by the KQML parser from Saci but, in other languages, developers must use an existent KQML parser or create a new one. An alternative solution to this problem is the inclusion of a CORBA interface to Saci's KQML parser, to be considered in future releases. Second, the CORBA bridge implementation will require changes if, in the future, the interfaces of Saci changes. If this occurs, it is necessary to go back to the IDL file and repeat all the implementation process, so this solution is more advisable to stable architectures, with minor changes. Finally, we have not tested this solution with ORBs from other suppliers. This is important, because even in a open standard like CORBA, incompatibilities can appear, consequently obfuscating the proposed interoperability.

References

[1] Booch, G., Rumbaugh, J., Jacobson, I. *The Unified Modeling Language User Guide*. Addison-Wesley Publishing Company, 1999.

[2] E. Gamma et al. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[3] Foundation for Intelligent Physical Agents. *FIPA Abstract Architecture Specification*. Geneva, Switzerland, 2000. http://www.fipa.org.

[4] Foundation for Intelligent Physical Agents. *FIPA ACL Message Structure Specification*. Geneva, Switzerland, 2000. http://www.fipa.org.

[5] Foundation for Intelligent Physical Agents. *FIPA Agent Message Transport Service Specification*. Geneva, Switzerland, 2000. http://www.fipa.org.

[6] Hübner, J.F., Sichman, J.S. Saci Programming Guide. Technical report, Universidade de São Paulo, 2000.
http://www.lti.pcs.usp.br/saci/doc/programmingGuide.pdf.

[7] Hübner, J.F., Sichman, J.S. SACI: Uma Ferramenta para Implementação e Monitoração da Comunicação entre Agentes. In *Proceedings of the International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI (Open Discussion Track)*, pages 47–56, São Carlos, 2000. ICMC/USP. http://www.lti.pcs.usp.br/saci.

[8] Labrou, Y., Finin, T. *A proposal for a new KQML specification.* UMBC, Baltimore, 1997.

[9] OMG. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, Inc., 1999.

[10] Orfali, R., Harkey, D., Edwards, J. *Instant CORBA.* John Wiley & Sons, Inc., 1997.

[11] Orfali, R., Harkey, D., Edwards, J. *Client/Server Programming with Java and CORBA.* John Wiley & Sons, Inc., 2nd. edition, 1998.

[12] Rohnert, H. The Proxy Design Pattern Revisited. In Vlissides, J.M.; Coplien, J.O; Kerth, N.L., editor, *Pattern Languages of Program Design*, chapter 7, pages 105–117. Addison-Wesley Publishing Company, 1995.

[13] Siegel, J. *CORBA 3 - Fundamentals and Programming.* John Wiley & Sons, Inc., 2nd. edition, 2000.