MobiliTools: An OMG Standards-Based Toolbox for Agent Mobility and Interoperability.

Bruno Dillenseger
France Télécom R&D (formerly known as Cnet), BP98, F-38243 Meylan Cedex, France

Abstract: One of the keys to success for applications of mobile and/or intelligent agents in large-scale open systems such as Internet is the ability of heterogeneous agents to cooperate and negotiate, and meet if they are mobile. This heterogeneity support requires the adoption of standards at the underlying distributed system level to support interoperability in agent management, mobile agent transport, and agent communication transport. This paper shows how both OMG standards and a modular architecture based on three kinds of component — agent mobility kernel, agent communication tools, and agent activity kernel — makes it possible to build a variety of heterogeneous mobile agent platforms with ad hoc features while preserving interoperability.

1 YET ANOTHER JAVA MOBILE AGENT PLATFORM?

1.1 A new paradigm for distributed systems

Classical techniques for distributed systems are based on client/server, code on demand, and remote evaluation paradigms, which finally result in moving code, and/or data, and/or control, as described in [14]. Now, mobile agents bring everything together into a new paradigm.

This paradigm has been introduced by Telescript [15] throw the *remote programming* concept, to reduce network load and latency, and to suit temporary network connectivity. As underlined in [9], there is little chance to find a “killer application” of mobile agents, but the paradigm is nice for any distributed application spread in a large-scale dynamic open system, where adaptation capability, through dynamic re-distribution of a set of cooperating agents, is a key to coping with changing hosts and network conditions, or to optimize the execution of distributed services.

But this nice anthropomorphic paradigm may not be so easy to handle practically. Besides security issues, which are critical to real large-scale applications, transparency, reliability, scalability and interoperability are other key challenges.

1.2 Limitations of today’s mobile agent platforms

1.2.1 Transparency

Today’s typical mobile agent platforms are built on a centralized programming language, enhanced with remote communication capabilities, and finally completed with mobility features (e.g. Java-based platforms). This final add-on of mobility deeply changes the behaviour of the original programming framework. For instance, many useful JDK packages are not designed for mobility, and transparency to mobility issues arise for any access to resources such as threads, files, sockets...

This is the reason why Java-based frameworks include specific models and tools for agent activity, communication and mobility, and specify programming restrictions. For instance, creating threads is discouraged (or forbidden) by Voyager [20] and Grasshopper [17], because the platform needs to tightly manage the agent activity. In some platforms, insufficient or disregarded restrictions can result in unspecified and indeterminist behaviour if an agent moves while it is involved in communication. As a matter of fact, communication has an impact on agent activity, and mobility has consequences on both communication and activity.

Full transparency would consist in having strong mobility as defined in [6], maintaining not only the agent state, but also the state of its activity and of its bindings to resources, including on-going communications.

1.2.2 Scalability

Both activity and communication models are of great importance for scalability. Java-based platforms that create (at least) one thread of activity per agent are examples of non-scalability if one imagines hundreds or thousands of agents needing to meet in one place.

Communication tools are also determining in scalability. Agents need to communicate locally, to take advantage of the remote programming paradigm, but also remotely, as explained in [13]. Remote communication may be implemented in a number of ways, with more or less state-of-the-art properties in terms of persistence, reliability, guaranty of delivery and causality ([2], [13]). Unfortunately, these outstanding properties typically rely on distributed algorithms introducing scalability limitations.

1.2.3 Interoperability

Last, but not least, it must also be considered that mobile agents’ specific properties are dedicated to large-scale, dynamic, open distributed systems (e.g.

Internet). In such a context, heterogeneous mobile agents need a common high-level communication language to understand each other, as well as a standardised distributed execution and communication infrastructure to interoperate.

FIPA's [16] and KQML-based Agent Communication Languages are emerging standards for making agents understand each other, negotiate and cooperate. But high-level communication also requires a lower level of interoperability, on the communication transport level. Unfortunately, no standard communication infrastructure actually emerges to transport messages between heterogeneous agents. Mobile agents also need to move around in a standardised infrastructure, dealing with a common conceptual framework.

Today's mobile agent platforms typically come with specific integrated frameworks making it difficult to introduce interoperability support. Nevertheless, Voyager's CORBA support and Grasshopper's MASIF compliance are encouraging effort examples towards interoperability.

1.3 MobiliTools' specific approach

MobiliTools is a set of CORBA-based Java tools for mobility that can be used separately. The specific architecture relies on two main principles:

1. a clear separation between object mobility support, communication tools, and activity management;
2. use of standard middleware for agent and communication transport.

Principle (1) is motivated by the idea that there is no universal mobile agent framework. It is preferable, instead, to create a number of interoperable agent frameworks by choosing and combining different communication tools and agent activity schemes, on top of a mobility kernel. For instance, if at least one of the communication tools is independent from the mobility kernel, it can be used by any other agent platform or software to interoperate.

Principle (2) enforces interoperability by choosing a standard communication layer, not only between agents, and between agent platforms, but also between agents and legacy applications. Moreover, communication middleware comes with useful generic services and tools meeting typical distributed systems' needs.

Mixing these two principles results in the architecture shown by Figure 1. Any component may be replaced or reused to build a variety of agent frameworks with a common support for agent and/or communication transport.

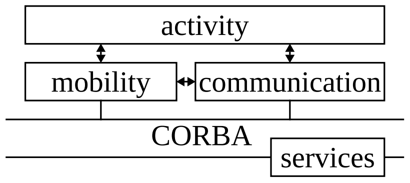

Figure 1: MobiliTools architecture

2 OMG STANDARDS AND AGENT TECHNOLOGY

2.1 Corba

OMG's Common Object Request Broker Architecture makes it possible for distributed programmes to perform remote calls on each other, regardless of their programming languages, in an object-oriented manner, while hiding network layers and operating systems heterogeneity. This standard is the result of a consortium grouping the major companies in information technology, and has several commercial and free implementations. CORBA support in some web browsers and in Java 2 is a sign of maturity.

CORBA comes with common services for distributed systems such as localisation (naming service, trader), and event-oriented communication (event service). Persistence, transactions, and security are also addressed. All these topics are of great interest for mobile agents, and everything can be re-used (as is, or as implementation "templates"), without enforcing any programming language (provided that the mapping exists from the Interface Definition Language to the target language), while relying on a well known, specified and widely available standard.

CORBA is an opportunity for interoperable basic management of agents, transport of mobile agents, and transport of agent communication. [3] describes several agent platforms developed on top of middleware such as CORBA. These platforms show in particular how several programming languages may co-exist to allow several programming levels, and how the middleware can be fully hidden to the agent programmer.

CORBA implementations do not actually support object mobility, but they can be used for every stationary component in a system of agents: execution environments hosting agents, infrastructure for agent communication, directory service...

2.2 Mobile Agent System Interoperability Facilities

OMG's first contribution to agent technology is the MASIF specification [10], dedicated to the interoperable management of agents and agent platforms. MASIF's framework is based on the following concepts: *Agents* autonomously act on behalf of a person or an organization called an *authority*. Agents are executed in *places,* hosted by *agent systems* (see Figure 2). *Mobile agents* have the ability to move from place to place, between agent systems, provided that their *agent system type* is recognized by the destination agent system. Agent systems are also bound to an authority, and may be grouped into a *region* if they are bound to the same authority. Agents are given a globally unique *name* resulting from the triplet {authority, agent identity, agent system type}.

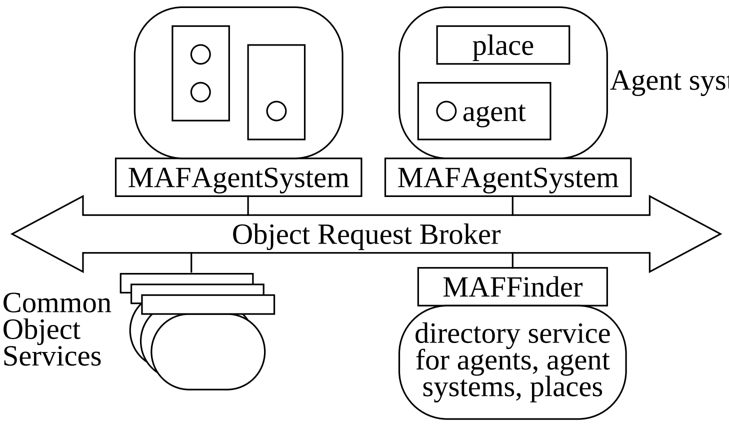

Figure 2: MASIF conceptual framework, with MAFFinder and MAFAgentSystem interfaces.

This framework is managed via two CORBA interfaces. Interface `MAFAgentSystem` must be implemented by agent systems to manage agents (create, suspend, resume, terminate), to receive migrating mobile agents, and to transfer agent classes. Interface `MAFFinder` is dedicated to registration and lookup of agents, places and agent systems.

2.3 CORBA 2.3, OMG Agent Working Group

OMG's interest in mobility and agent technology is growing. CORBA 2.3 specifications are contributing to object mobility support by including an object-by-value feature that makes it possible to pass programming language objects as invocation parameters.

As far as agent technology itself is concerned, MASIF is only a preliminary step in OMG's work. The Agent Working Group (AWG) [19] was created at the end of 1998, in order to open a forum for educating OMG in agent technology, and develop an architectural framework supporting agent technology in a compatible and complementary way with OMG's specifications. The AWG is also interested in coordinating standardisation work with other consortia in the agent field, such as FIPA.

The AWG started to write an "Agent technology green paper" [11], issued a Request For Information on "Agent technology and Object Management Architecture" in 1999, and is currently working on an "Agent Technology White Paper and RFP Roadmap" [12]. RFPs will focus on interoperability, agent communication language, security, mobility, as well as distribution, robustness and scalability.

3 THE AGENT COMMUNICATION TRANSPORT SERVICE

3.1 Overview

The Agent Communication Transport Service (ACTS) is a CORBA service for transporting messages between heterogeneous agents, whatever mobile or not, CORBA objects or not. Accordingly to the decomposition suggested in Section 1.3, the ACTS is a communication tool, independent from both the mobility kernel and the activity model. Although it is independent from MASIF, the ACTS may be considered as a complement enabling interoperability between agents for remote communication, through the definition of extra interfaces. A detailed description of the ACTS can be found in [4]; we present the basics below, and then compare the ACTS with other related work.

3.1.1 How it works

The ACTS is based on one or several servers, playing the role of *message port factory*. Basically, *message ports* are stationary FIFO buffers where agents can add and retrieve messages of CORBA “Any” type. Note that agents need not be CORBA objects. A message port can be switched from this default *store mode* to *forward mode*, by declaring a *message port listener*. A listener is a CORBA object that receives pending and incoming messages. This listener may be invalidated, either explicitly, or as soon as a CORBA communication failure occurs with this object. Such a communication failure may spring from a loss of network connectivity with the listener, or may be caused by an obsolete CORBA object reference due to the listener mobility. No message is lost, and the FIFO order is maintained anyway.

3.1.2 Typical ACTS usage

The ACTS may be distributed on a number of servers running on well connected nodes (ACTS servers can be considered as e-mail servers). An agent may have one or several message ports in different network areas in order to improve communication performance and/or reliability. According to its specific constraints, an agent may choose either a pure asynchronous communication model, where it polls its message port (store mode), or a more "reactive" model where it gets incoming messages on the fly (forward mode). In the latter case, the new reference of the listener has to be registered after each move in order to keep the "reactive" behaviour. Note that the forward mode should be handled very carefully, since each forwarded message creates a thread of activity in the listener.

3.1.3 Customization: ACTS personalities

The ACTS *personalities* hide the CORBA infrastructure and the ACTS interfaces, while providing easy-to-use communication utilities for Java. ACTS personalities also come with enhanced transparency support, advanced communication features, and higher level addressing.

The ACTS *Mailbox personality* wraps message ports into Mailbox Java objects. Mailboxes are designated with high-level addresses, consistent with MASIF's region concept (`agent_name@region_name`). Multicast and unicast features are supported by addresses transparently targeting a group of mailboxes in a given region (`group_name@region_name`). The CORBA naming service is used to register and find the ACTS servers and the mailboxes' message ports:

- name "/MAF/*region_name*/acts/factory" for ACTS servers;
- name "/MAF/*region_name*/acts/mailbox/*mailbox_name*" for message ports bound to ordinary mailboxes, or arbitrary unique names in naming context "/MAF/*region_name*/acts/mailbox/*mailbox_name*/" for message ports bound to group mailboxes.

Section 4.3 details the specific naming service usage for scalability.

The ACTS *Logged Mailbox personality* is a Mailbox extension providing the programmer with communication tracing tools and event ordering based on a Lamport Clock mechanism [8]. The ACTS *FIPA personality* is a FIPA-oriented use of the ACTS, compliant with FIPA'98 specifications for Agent Management and agent-agent interactions [5].

3.2 Further discussion on the ACTS and communication issues

3.2.1 Agent communication schemes

[1] identifies two communication schemes in agent-based systems: agent-to-agent communication where partners are addressed by globally unique identifiers, and anonymous communications where partners do not know each other (event model). Through the Mailbox personality and its multicast/unicast enhancement, we see that the ACTS supports both schemes, both in forward and store mode. Another way to achieve this is to mix the message ports with the CORBA event service, but the event service can't be used directly by agents because of their mobility.

3.2.2 Communication delivery

Three basic techniques can be used (and mixed) to reach a moving destination:

1. use a directory which binds constant names to changing locations;
2. broadcast;
3. replace the mobile agent by a forwarding “ghost” on each move;

Technique (1) is often criticized for it relies on a centralized service. Nevertheless, this technique is currently of common use in mobile phone. Applicability domain of technique (2) is typically the LAN, where broadcasting does not necessarily generate extra messages (e.g. Ethernet). Larger-scale broadcast is a problem since it typically consumes too much network bandwidth and processing time in all the recipients (and/or in any intermediate communication element). Technique (3) comes with risks of reference chain breaking and forwarders proliferation. Moreover, it can not be applied when the reason for mobility is a node or network link shutdown.

All these techniques can be defeated in the case of highly mobile agents because messages may be routed permanently and never reach their destination. [13] presents a solution derived from the distributed snapshot algorithm. It is based on a synchronisation between message propagation and moving agents on communication links. However, this work needs to be continued in order to take network and node faults into account, and scalability is likely to be a problem.

The ACTS approach is different: an agent is always addressed by other agents through a single reference that never changes (the message port). The only reference that needs to be updated is the reference to the listener when a message port is operated in forward mode. Doing this update is of the agent’s responsibility. In the special case of a highly mobile agent, it is recommended not to use the forward mode, not because messages could be lost, but because messages might never reach the moving listener. The store mode seems to be the right communication model in this case.

4 THE SIMPLE MASIF IMPLEMENTATION

4.1 SMI overview

Accordingly to the decomposition of agent platforms given in Section 1.3, SMI implements a mobility kernel in Java. Starting from MASIF specification, SMI aims at providing a generic, light-weight and well-specified environment for mobile Java objects.

4.1.1 Agencies

An agency is an execution environment for mobile agents, called agent system in MASIF's terminology. Basically, they are instances of class `Agency` running in a Java Virtual Machine. Each agency belongs to a region, has a name (unique in the given region), and is bound to an authority. An agency is also a CORBA server implementing MASIF's `MAFAgentSystem` interface. Its CORBA object reference is registered in the naming service (see Section 4.3).

Agents can be managed through the `MAFAgentSystem` interface and methods of class `Agency`. Operations include creating and terminating an agent, suspending and resuming an agent activity, moving an agent, listing the names of hosted agents, and getting information on a local agent.

4.1.2 Mobile objects/agents

Agencies have methods for creating and managing any Java object implementing the `MobileObject` interface. This interface mainly consists of call-backs related to the lifecycle of mobile agents (see Section 4.2). `MobileObject` implementations also have to implement the `java.io.Serializable` interface since Java serialization is used to generate mobile agents' states. As specified by MASIF, an agent resides in a place, and has a unique name combining an identity, an authority and an Agent System Type identifier.

4.2 MobileObject lifecycle

The design of interface `MobileObject` is a straightforward mapping of the MASIF framework: agents may be created, moved, suspended, resumed and terminated. Agents have to be informed when such lifecycle events start, succeed or fail (see Table 1), not only to properly react, but also to be able to

Table 1: Agent lifecycle management and `MobileObject` interface.

Agency method	involved MobileObject call-back(s)
`createAgent`	`afterBirth`
`resumeAgent`	`resume`
`suspendAgent`	`suspend`
`moveAgent`	`beforeMove afterMove afterMoveFailed`
`terminateAgent`	`beforeDeath`

deny permission: an agent can refuse creation, mobility, or reinstallation after a move, by throwing an exception in the corresponding call-back.

For instance, method `moveAgent()` of class `Agency` involves a number of steps which can fail for various reasons: the specified agent or the destination agency doesn't exist, the destination agency can't be reached because of a communication problem (network, CORBA, naming service), or agent de/serialization has failed. But the agent may also abort the move by throwing an exception before (in `beforeMove()`) or during serialization, during or after (in `afterMove()`) deserialization. If the move is aborted after the serialisation step, the `afterMoveFailed()` call-back is invoked.

4.3 Naming service distributed exploitation

SMI agencies are bound to unique names in the CORBA naming service, according to a naming scheme extending MASIF's concept of region: "/MAF/*region_name*/agency/*agency_name*". As a result, agencies (like mailboxes' message ports and ACTS servers, see Section 3.1) can be found via high-level deterministic names, helping region interconnection.

A specific naming service administration is required to avoid a bottleneck effect. The first idea is to distribute the naming service on several servers, with one name server per region. Each name server contains the name bindings for its own region, and is federated with the other name servers in the "/MAF/" naming context. As a result, resolution of name "/MAF/regionA/..." with region B's name server is transparently forwarded to region A's name server. To go further on distribution, region names may contain sub-regions (e.g. "regionA/sub-region1/..."). In this case, one name server can be responsible for each sub-region. Note that this distribution also applies for the ACTS servers.

4.4 Back to MASIF and interoperability

MASIF specifications practically supports interoperability for basic agent management tasks, through the definition of:
– a common framework of places, agent systems, region, etc.;
– a service for agent, place and agent system registration and lookup;
– an external interface for agent lifecycle.

All these points don't require a smart interpretation, and their implementation is quite straightforward. But interoperability is not fully specified for agent mobility, and is not addressed at all for agent communication. Since the latter issue is explicitly not in the scope of MASIF (the ACTS described in Section 3 suggests one solution), let's focus on the former issue. MASIF's mobility support is based on two operations:
– `receive_agent()` is invoked on the destination agency to transfer an agent — parameters include the agent profile, the agent state, the agent class

name, and a CORBA object reference to the agent system providing the agent's classes;
– `fetch_class()` is invoked by the destination agency on the class provider to get the incoming agent's locally undefined classes.

4.4.1 Agent profile

Heterogeneity management is based on the provisioning of an *agent profile*. A profile contains a set of identifiers specifying the agent programming language, the agent system type, versioning information, and serialization format. Identifiers are already defined for Java, Tcl, Scheme, Perl, Aglets, MOA, AgentTcl and Java object serialization.

SMI naturally gets the Java language and the Java object serialization identifiers, and is given a free identifier for "SMI agent system type". SMI's policy is to reject agents of any other agent system type trying to move in. Since a dedicated exception is missing, the generic `MAFExtendedException` exception is thrown. It could be imagined that hosting an agent of a different agent system type but of the same programming language could be easy, especially in the case of Java. But several implementation choices remain about de/serialization, class loading and agent lifecycle hooks. Let's discuss the interoperability issues in the case of Java as a common programming language (in the case of heterogeneous programming languages, we imagine a pseudo-agent system switching agents on to the right agent system).

4.4.2 Agent deserialization and classloader

Using standard Java object serialization does not mean that a standard `ObjectInputStream` can be used for deserialization. A specific classloader must be provided for each agent in order to fetch missing classes from the specified class provider, using the specified codebase, for the specified agent profile. This classloader must be supplied by a specific `ObjectInputStream` deserializing the agent state.

There are several other implementation choices about class loading issues, which may lead to non-interoperability. For instance, the classloader used for agent deserialization may be quite different if it assumes that classes are transferred as a whole as a parameter of `receive_agent()`, or downloaded on the fly from the class provider if they are locally undefined.

Missing classes in the destination agent system may be fetched either always from the same agent system, or from the source agent system. The former technique introduces a serious bottleneck, and may prevent an agent from moving from agency B to agency C if the class providing agency A is unreachable. The latter technique may cause a proliferation of classes, since it requires that the agencies keep byte code for hosted agents' classes. The main issue is scalability, since the amount of byte code stored in each agency may rapidly grow. SMI uses this technique however, because it results in a much more fault tolerant overall distribution. This has to be tuned and refined, but detecting and discarding useless classes is complicated by Java's reflective features.

4.4.3 "Internal" interfaces

Finally, the main difficulty for interoperability within a given programming language, is that standard hooks must be specified to tell the agent it is going to move or die, or it has just moved, or it has just been born... A common lifecycle interface such as SMI's `MobileObject` (see Section 4.2) should be defined for each language.

Local interactions with the agency and the other agents also need to be specified. For instance, an agent willing to move must be given a standard way to request the move from the agent system it is residing in. Then, supporting the remote programming paradigm for heterogeneous agents requires a standard mechanism to initiate and handle a local communication tool through a standard interface.

4.5 To be added: agent activity models

For the sake of genericity, SMI does not enforce any agent execution model. Agents are responsible for starting, suspending, resuming and terminating their activity accordingly to the corresponding lifecycle call-backs. Agents may launch a thread of activity, or share a pool of threads. The former approach fully supports autonomous agent activity, but is not scalable, while the latter approach is essentially dedicated to event-driven agents, like in [2].

Event-driven activity may be implemented using the reactive programming model. Such a model consists in splitting execution into logical time slices, or *instants*. Reactive objects react to events, combinations of events, or absence of events, and generate events that are consumed in the same instant. An instant ends when all events are consumed, and a new instant starts when new external events appear.

The benefit of such an approach is that between two instants, the state of an agent is stable and very well defined. Then, move requests can be transparently executed after the end of each instant, without affecting the programming model. Moreover, work described in [7] has produced a Java prototype able to run thousands of reactive objects, which is a promising performance regarding scalability concerns.

5 CONCLUSION

Through the presentation of MobiliTools, this paper practically explores:

– the applicability of OMG standards for making interoperable mobile agent platforms;
– how a mobile agent platform can be built as a combination of a mobility kernel, communication tools, and agent activity support.

Although MASIF brings limited interoperability support, mainly because of the "internal interfaces" issue, it is an interesting starting point for the architecture of mobile agent platforms. CORBA is convenient to implement the stationary parts of the global infrastructure, responsible for transporting and managing agents and messages. The naming service, used in an appropriate manner, provides a scalable directory for high-level location-independent references.

At last, the approach based on the assembly of independent components improves comprehensibility of transparency issues, and leads to a variety of interoperable combinations suited to various needs. For instance, the ACTS may be used in any mobile agent platform without any other MobiliTools. In the same way, SMI may host any agent activity and communication framework while managing mobility through the `MobileObject` interface.

Next steps include tuning and completion in order to fully implement MASIF, enhance the communication support, and offer a couple of agent activity models. Strong mobility support is on the way, on the basis of the reactive programming model.

ACKNOWLEDGEMENTS

The Agent Communication Transport Service has been developed for the MIAMI project [18], with design contribution from Stefan Covaci and Alexander Yip. The FIPA personality has been made by Huan Tran Viet. The logged mailbox personality has been made by Anne-Marie Tagant.

REFERENCES

1. J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, M. Strasser: Communication concepts for Mobile Agent Systems. In mobile Agents: 1st International Workshop MA'97, Lecture Notes in Computer Science, April 1997, Springer, pp. 123-135.
2. L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann, S. Lacourte: An Agent Platform for Reliable Asynchronous Distributed Programming. Symposium on Reliable Distributed Systems (SRDS'99), Lausanne (Switzerland), 20-22 October 1999.

3. B. Dillenseger: From Interoperability to Cooperation: Building Intelligent Agents on Middleware. Lecture Notes in Artificial Intelligence 1437 (Proc. of IATA'98), Sahin Albayrak, Francisco J. Garijo Eds. Springer 1998, pp. 220-232.
4. B. Dillenseger, Huan Tran Viet: Towards full agent interoperability. In Proc. of 2nd International ACTS Workshop on Advanced Services in Fixed and Mobile Telecommunications Networks. 9-10 September 1999, Center for Wireless Communications, Singapore.
5. FIPA 98 Specification. Foundation for Intelligent Physical Agents (Geneva, Switzerland),1998. see [16]
6. A. Fugetta, G. P. Picco, G Vigna: Understanding code mobility. IEEE Transactions on Software Engineering, vol. 24, No 5 (1998), pp. 342-361.
7. L. Hazard, J.-F. Susini, F. Boussinot: The Junior Reactive Kernel. Rapport de recherche No 3732, July 1999, INRIA Sophia Antipolis (France).
8. L. Lamport: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, July 1978, Vol. 21, No 7., pp. 558-565.
9. Danny B. Lange, Mitsuro Oshima: Seven good reasons for mobile agents. Communications of the ACM, Vol.42, No 3, March 1999, pp. 88-89.
10. Mobile Agent System Interoperability Facilities Specification. Joint submision: GMD Fokus & IBM Corp., supported by Crystaliz Inc., General Magic Inc., The Open Group. OMG TC document orbos/97-10-05 (1997).
11. OMG Agent Working Group: Agent Technology Green Paper. Document ec/99-12-02, 24 December 1999. see [19]
12. OMG Agent Working Group: Agent Technology White Paper and RFP Roadmap. Ref. internet/99-11-01, draft .02, 29 November 1999. see [19]
13. Amy L. Murphy, Gian Pietro Picco: Reliable communication for highly mobile agents. In proc. 1st International Symposium on Agent Systems and Applications, 3rd International Symposium on Mobile Agents, Palm Springs (USA), D.S. Milojicic ed., october 1999, IEEE Computer Society, pp. 141-150.
14. J. Vitek: New paradigms for distributed programming. In proceedings European Research Seminar in Advanced Distributed Systems, Zinal (Switzerland), march 17-21, 1997.
15. J. White: Telescript technology: the foundation for the electronic market place. General Magic White Paper, General Magic, 1994.

WEB REFERENCES

16. FIPA - http://www.fipa.org/
17. Grasshopper - http://www.ikv.de/
18. MIAMI - http://www.fokus.gmd.de/research/cc/ecco/miami/
19. OMG Agent Working Group - http://www.objs.com/isig/agents.html
20. Voyager - http://www.objectspace.com/