The Agent Pattern: A Design Pattern for Dynamic and Distributed Applications

Alberto Silva, José Delgado
{Alberto.Silva, Jose.Delgado}@inesc.pt
INESC & IST Technical University of Lisbon,
Rua Alves Redol, nº 9, 1000 LISBOA, PORTUGAL

Abstract

This paper presents the Agent pattern, a design pattern useful to develop dynamic and distributed applications. The Agent pattern provides a clean and easy way to develop agent-based applications, mainly in open and large-scale distributed environments such as the Internet and application areas such as Electronic Commerce. The Agent pattern encapsulates a business specific class (a specialization of the `Agent` class), with some user identification and a specific security policy, providing distribution, security and persistence transparency. Furthermore, this paper presents a detailed application of this pattern in the AgentSpace framework, as well as a brief application to the Telescript and Aglets Workbench.

1 Intent

To define autonomous active objects to easily build dynamic and distributed applications. These active objects are called agents because they are autonomous and execute specific tasks on behalf of their users.

2 Scope and Motivation

Maybe one of the most understandable agent definition is due to Wooldridge and Jennings [WJ95], which discussed agents following two basic notions: a weak and a strong one. The strong notion involves Artificial Intelligent techniques and models to characterize agents using *mentalistics* notions, such as knowledge, beliefs, intentions and obligations, or even with emotional attributes. Based in that tentative agent definition, the discussed agent pattern in this paper focuses on this weak notion where autonomy, sociability, reactivity and even mobility are the most important characteristics. Additionally, it is important to state that the focus of the proposed agent pattern is more architectural-oriented than application-oriented. This means that the focus is based on low-level issues such as synchronization, thread management, communication, security handling or persistence. On the other hand, the focus of application-oriented approaches, such as those found in the AI field, are mostly on knowledge representation, cooperation/collaboration definition models, agent communication in a heterogeneous and high level scale, etc.

Other aspect that should be clarified is the relationship between agent support systems (ASSs), and object request brokers (ORB), such as CORBA implementations, RMI, DCOM. Both systems present some similarities because both of them intend to support distributed applications. However, ASSs provide a framework to develop applications based mainly on the agent paradigm, whereas ORBs on the object-oriented paradigm. Nevertheless, an agent is at some scale an active object. So, in our opinion, the main difference between both types of systems is better discussed following two basic vectors: flexibility and specificity. ASSs are designed and implemented to support specifically a restricted number of application-families. On the other hand, ORBs are more flexible, application-family independent and consequently they present low-level support. We expect that the development of the next generation ASSs would be developed on top of some kind of ORB, although the contrary would not be true.

For motivation let us consider an electronic commerce application in a vast and open environment such as the Internet and three basic entities: the buyer, the salesman and the broker. In an agent-based approach, users interact with their own agents and delegate their specific tasks. Agents interact between themselves in order to perform these respective tasks. Figure 1 shows a scheme with the interaction between agents and between agents and their users.

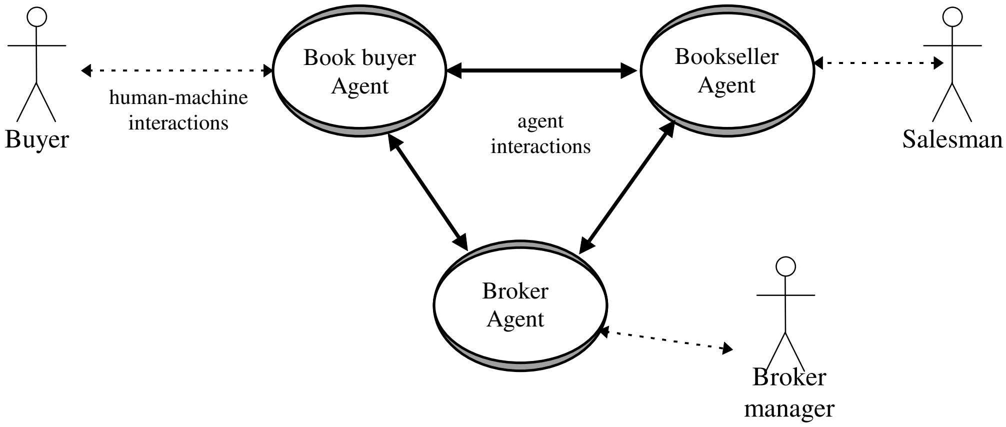

Figure 1: Global view of an agent-based application.

For example, for a user to buy a book he should only interact with his/her agent in order to give it some specific information. For instance: the book related information (e.g., title, author names, etc.); the maximum supported price; the initial and minimum price to start eventual negotiation; the address of some public booksellers' broker. All the subsequent actions – such as: accessing and interacting with the broker; obtaining the addresses of the relevant bookseller agents; accessing and eventually negotiating with some bookseller agent; deciding whether to buy or not; buying the book; etc. – are executed indirectly by the respective agent. Eventually, users will monitor the current execution state of their agents. Figure 2 shows a UML [Rat97] scenario diagram corresponding to the described operation.

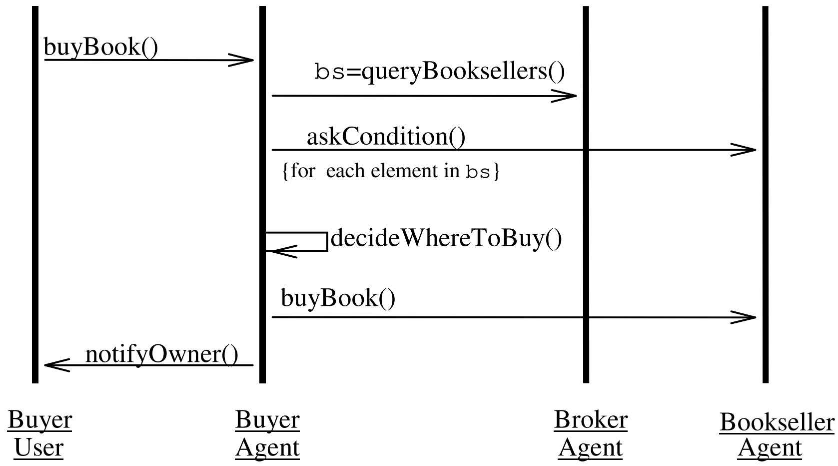

Figure 2: Buying operation in an agent-based application – scenario diagram.

As suggested in this example, agents may be viewed as a new interface paradigm to help the end-user in accessing this new class of Internet applications, including electronic commerce applications. In this world, the end-users change the way they interact with the computer, from *direct manipulation* (e.g., word processors, Web browsers, and so on) to *indirect management* (e.g., Web information search). Using agents, users can delegate a set of tasks to be done by agents instead of themselves. This new paradigm is especially attractive to help the users in complex, tedious or repetitive tasks and in open, dynamic, vast and unstructured information sources such as those found in the Internet.

We define an *agent-based application* (or ABA for short) as a dynamic, potentially large-scale distributed application in an open and heterogeneous context such as the Internet. The basic conceptual unit for designing and building ABAs is the agent as defined above.

The notion of ABA is quite novel. An ABA is not a typical application that is owned and managed by some person or some organization. Instead, an ABA is best understood as a web of agents, each owned and managed by a number of entities with different (and possibly conflicting) goals and attitudes, hosted in different computing platforms, such as workstations and mobile phones.

ABAs have a number of characteristics and requirements that have been dealt with independently in the past. It is their interaction that poses problems. ABAs should be autonomous, heterogeneous, open, dynamic, robust and secure. All these characteristics make ABAs potentially very difficult to implement and use. However, we believe that agent support systems, and in particular AgentSpace (see Section 7), will help developers to build and to manage them.

3 Applicability

Use the Agent pattern when someone wants:

- To define autonomous active-objects supported by some framework.
- Clients to be able to ignore low-level details, such as distribution or security aspects, for instance to obtain references to agents or to interact with agents transparently.
- To develop and to manage dynamic and distributed applications in an easy way.

4 Structure and Participants

Figure 3 shows the Agent pattern represented by a UML [Rat97] collaboration diagram, its main participants and involved collaborations.

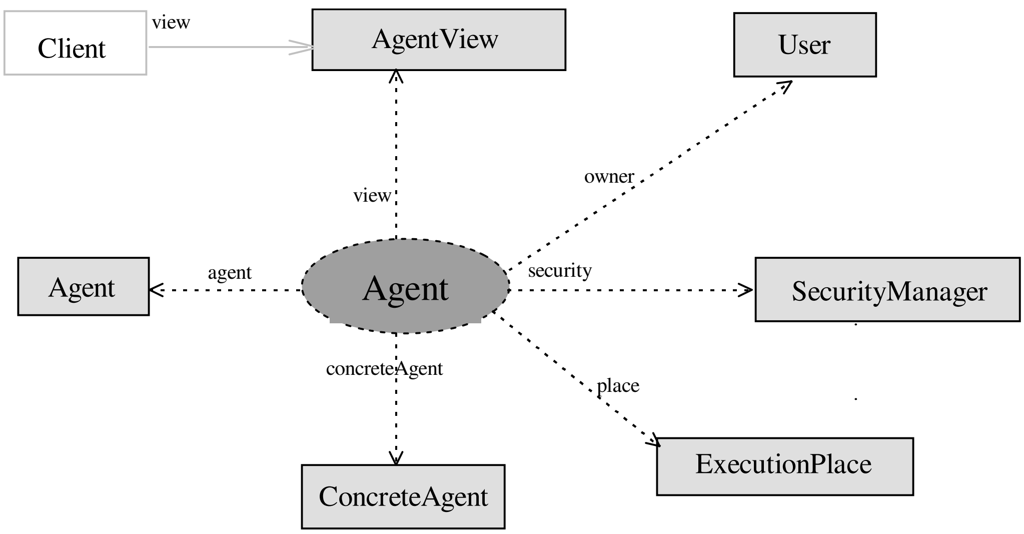

Figure 3: Generic structure of the Agent pattern – collaboration diagram.

Figure 4 shows the generic static structure corresponding to the previous depicted collaboration diagram.

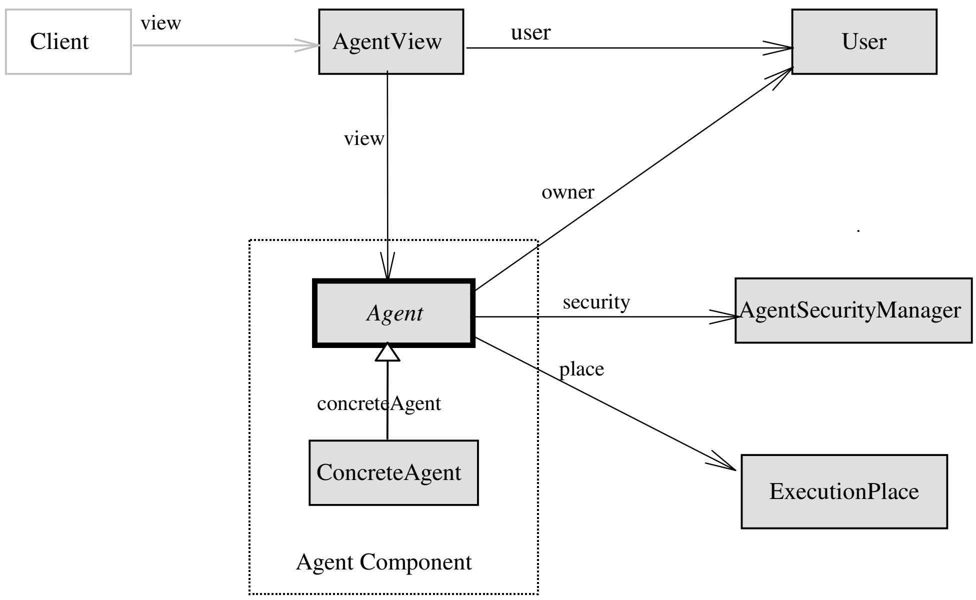

Figure 4: Generic structure of the Agent pattern – class diagram.

The main participants in the pattern are:

- **Client**
  - Manipulates agents through the `AgentView` reference.
  - Clients can be other agents or other objects, for instance Java applets.
- **AgentView**
  - The `AgentView` is an adaptation of the *Proxy* [GHJV95] and *Remote Proxy* [BM+96] patterns. This pattern is very suitable to support transparent and secure access to these different types of objects.
  - The aim of `AgentView` is to provide transparent access to agents. This access is done indirectly through proxies in order to protect them, and to hide transparently their current localization (this is important due to the mobility characteristic). Additionally, `AgentView` avoids the need to create and manage remote/virtual classes (e.g., stubs and skeletons in RMI and CORBA implementations). Examples of operations provided/protected through agent proxies are: `sendMessage`, `getCurrentPlace`, `start`, `moveTo`, `getClassName`, etc.
- **User**
  - The user is identified by a unique identity, which contains: his/her name; a public key; a set of certificates; the organization and country he/she belongs to; and his/her e-mail. In spite of all these attributes, the name is the unique mandatory field (however, this requirement is obviously application specific). Moreover, the user can have different identifiers depending on the context he/she belongs to. This specific identity, managed in every Agent Server's context, is represented by the `User` class, which contains, in addition to all fields mentioned above, the authentication attributes (e.g., login and password).
  - The agent's owner has necessarily an associated user identity, represented always by an `User` instance.
  - Different users, through the corresponding AgentView instance (see Figure 6), can access the same agent. Depending on the agent's security manager, each access is allowed or not (see Figure 7).
- **Agent**
  - The `Agent` abstract class is the visible and extensible part of the Agent pattern.
  - Basically, programmers should derive the `Agent` abstract class in order to build their own concrete classes. The agent class has three main groups of methods: (1) public final; (2) callbacks; and (3) helper methods, as depicted in Figure 5.

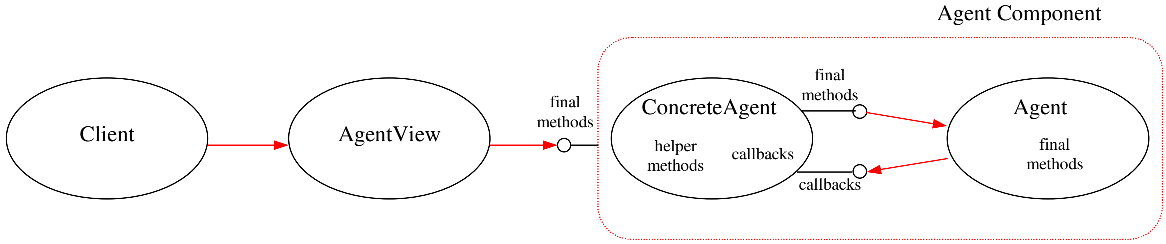

Figure 5: Agent's main groups of methods.

    - *Final methods* are pre-defined operations provided by all agent classes that cannot be changed by the programmer. Examples of these final methods are: `moveTo`, `save`, `die`, `backHome`, `clone`, `getId`, `sendMessage`, etc.
    - On the other hand, *callbacks* are methods customized by specific agent classes, and are usually invoked transparently as the result of some event. Events are trigged by some action started by the agent itself or by other related entity, such as an other agent, an end-user (via same applet), a time service, etc. The callback mechanism provides the desired extensibility of the Agent pattern. Examples of callbacks are: `run`, `onCreation`, `beforeDie`, `handleMessage`, etc.
    - Finally, agent classes also have *helper methods*, generally with private or protected access modifiers, in order to support specific functions of that class/object. These methods are used internally by callback methods. Examples are `atBookseller`, `decideWhereToBuy`, etc.
  - The `Agent` instance provides transparently several services, such as: persistence, communication, mobility, naming and access control. Additionally, the `Agent` instance should keep related information, such as: the current and native place identities, security policy object, a reference to the concrete agent itself, its own identity, its owner identity, a reference to the involved security manager, and the group of threads involved.
- **ConcreteAgent**
  - Concrete agent classes are `Agent` subclasses.
  - Basically they define helper methods and specialized callbacks, that, as a whole, they implements the agent's specific functionality.
- **SecurityManager**
  - This class specifies the agent access security policy.
  - The agent's `SecurityManager` instance controls all the operations made available on the agent component through each `AgentView` instance.
- **ExecutionPlace**
  - This class specifies the agent's computational environment, which corresponds to the place where it was created as well as where it is currently resident.
  - This class offers specific functions provided by the involved agent support system.

Figure 6 shows an example of a UML object diagram of the Agent pattern at run-time. There are just one agent component (corresponding to the `ca` object) and two client components (i.e., `c1` and `c2` objects) interacting with it. The figure shows that all the involved agent and client objects are associated with different users, respectively `owner`, `u1` and `u2`. Additionally, the locality of each component is not relevant, because this aspect is handled transparently by the `AgentView` class, and of course by the involved ASS.

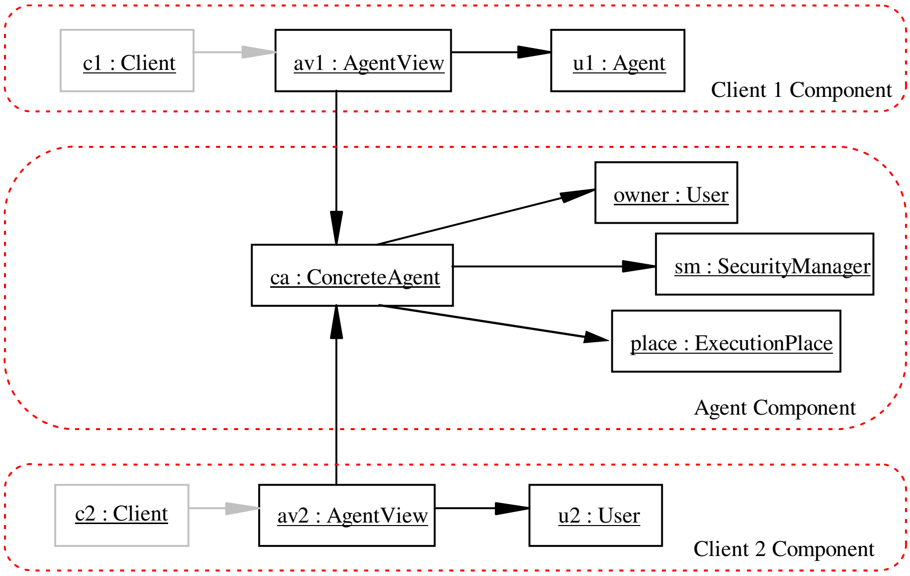

Figure 6: Application example of using the Agent pattern – object diagram.

5 Collaborations

Clients call standard agent operations through an `AgentView` instance. Depending on the agent's security policy and on the involved user, the operation is executed, or not, on the related agent instance.

Final methods are basically executed by the `Agent` instance. On the other hand, callbacks, resulting from the execution of final methods (e.g., `moveTo`, `die`, `sendMessage`), are executed by the concrete agent instance. Lastly, some helper methods may be invoked by the execution of some callbacks.

Figure 7 shows an abstract UML collaboration scenario of the Agent pattern.

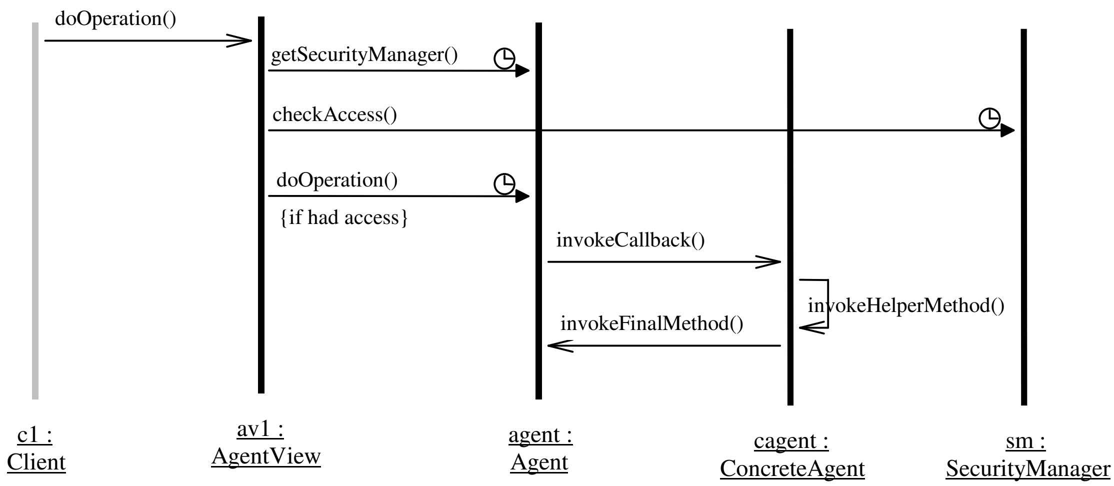

Figure 7: Generic interaction of the Agent pattern – scenario diagram.

6 Consequences

The Agent pattern introduces several benefits in designing and building agent-based applications, mainly in open, dynamic and distributed contexts such as the Internet. However, at the same time, its adoption raises some challenges and difficulties.

Benefits:

- Easy development of dynamic and distributed applications
  - Programmers don't have to manipulate remote accesses to objects/classes, such as the stubs and skeletons in Sun's RMI or the virtual objects in ObjectSpace's Voyager. They just need to know, or to query dynamically, (1) the agent identity; and (2) the agent's class interface (i.e., its set of messages/public methods).
- Simple integration with definition and management of users
  - The user management is handled implicitly by the system: (1) each agent/place has necessarily, and at least, one identified owner; and (2) other users can access and interact with a given agent, directly (e.g., via a specific applet) or indirectly (e.g., via other agent).
- Flexible definition of agent's security policy
  - The agent's security policy (as well as the agent's owner definition) is not statically coupled to the agent class definition. This association is just performed dynamically at agent creation time. This fact implies that instances of the same agent class can have different security policies (as well as different owners).

Problems/Challenges:

- The inexperience of users with the indirect human-machine interaction (delegation) paradigm.
- Security and privacy, namely in open and distributed contexts.
- Requires applications with novel business models.
- Requires the existence of some agent support system or framework.
- Requires the existence of generic clients to monitor and to manage the current execution and state of agents.

7 Implementation

7.1 Architectural Issues

The use of the Agent pattern requires an agent support system, which involves several architectural issues, such as:

- Identifiers: how to identify the related resources, such as the agent support system, places and even agents;
- Agent programming languages: in what languages should programmers develop agent classes;
- Agent communication languages: in what languages should agents communicate: open and independent/standard languages, such as KQML, IDL, XML, EDI; or in application-based structures and specifications;
- Security and access control: namely in open and distributed environments, and with mobile agents, this issue becomes very important;
- Low-level distribution support mechanism: what low-level mechanism should be provided so that agents can communicate with each other and move themselves from place to place;
- Persistence: in order to provide persistence to agents, because they must recover from failures and they must survive down periods of agent server where they may be currently hosted;
- Mobility: agents can navigate through a (static or dynamic) set of places in order to communicate locally with other agents/services or to meet at some defined public place. There are several technical issues raised due to mobility, such as: state maintenance; security; data and code closure; etc.
- Mobility and communication: how to handle communication between mobile agents, i.e., how can an agent communicate with an other one, if it doesn't know the current place of its partner.

7.2 AgentSpace: an Agent-based Framework

AgentSpace [SMD97, SMD98] is an implementation of a next-generation Java mobile agent system recently developed. The AgentSpace first version is now ready to be used by other programmers. Amongst other applications, AgentSpace is being used to implement a generic „Agent Manager Tool“ (with a graphical, AWT-based, user interface), and a prototypical electronic payment system based on SET [RM98]. In the near future AgentSpace will be used in the COSMOS project [COS97] that will build a framework to negotiate contracts on the Internet. It is available on the Internet for downloading at: http://berlin.inesc.pt/~agentspc.

AgentSpace was conceived to support, develop and manage ABAs. Figure 8 shows the three main components of AgentSpace.

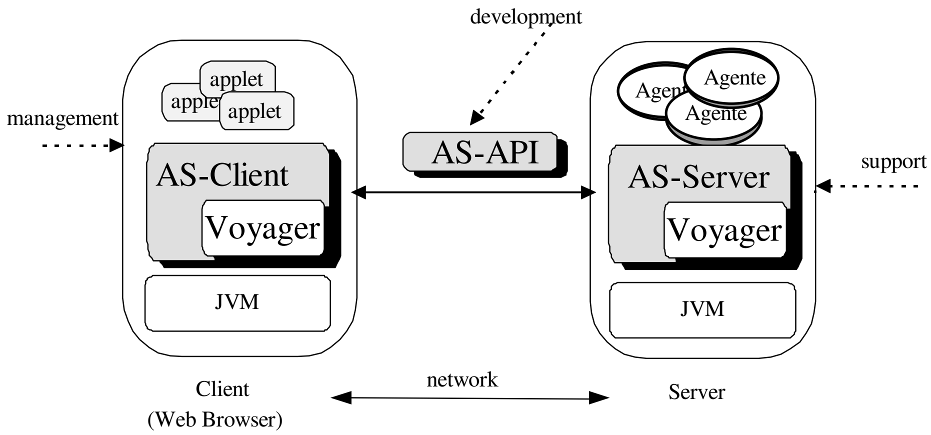

Figure 8: AgentSpace main components.

- *AgentSpace server* (*AS-Server*): is a Java multithreaded process in which agents can be executed. The AS-Server provides several services, namely: (1) agent and place creation; (2) agent execution; (3) access control; (4) agent persistency; (5) agent mobility; (6) generation of unique identities; (7) support for agent communication; and (8) optionally a simple interface to manage/monitor itself.
- *AgentSpace client* (*AS-Client*): manages and monitors (depending on the corresponding user access level) agents and related resources. The AS-Client is a set of Java applets stored in a AS-Server's machine in order to provide an adequate integration with the Web, offering Internet users the possibility to easily manage their own agents remotely. The AS-Client is able to access several AS-Servers, providing a convenient trade-off between integration and independence between these two components.
- *AgentSpace application programming interface* (*AS-API*): is a package of Java interfaces and classes that defines the rules to build agents. In particular, the AS-API supports the programmer when building: (1) agent classes and their instances (agents) that are created and stored in the AS-Server's database for later use; and (2) client applets (that are stored in the AS-Server's file system or even in the AS-Server's data base) in order to provide end-user interface to remote agents.

Both server and client components run on top of Voyager [Obj97] and Java Virtual Machine (JVM), and they can execute in the same or in different machines. Agents run always on some AS-Server's context. On the other hand, they interact with their end-user through (specific or generic) applets running in some Web browser's context.

Figure 9 shows, as a UML class diagram, the relationships between several related objects that are supported, developed and managed by AgentSpace: contexts, places, agents, users, groups of users, permissions, ACLs (access control lists), security managers, tickets, messages, and identities.

Each AS-Server is represented by one *context*, which is the most important and critical object of the AS-Server. The context contains the major data structures and code to support the AS-Server, such as lists of places, users, groups of users, meta-agent classes and access control lists.

Each context has a number of places. The *execution place*, or simply *place*, has mainly two objectives. First, to provide a conceptual and programming metaphore where agents are executed and meet other agents. Second, to provide a consistent way to define and control access levels, and to control computational resources.

The *agent* is the basic element of the system. Agents are identified by a unique and global identity. Agents are active objects that execute in some AS-Server, but from a conceptual perspective, they are currently in some place. Agents can navigate to other (local or remote) place if they have permission to do it. Just one user owns each agent. Nevertheless, other users (or even agents from other users) can interact with it, if allowed by the agent's security policy.

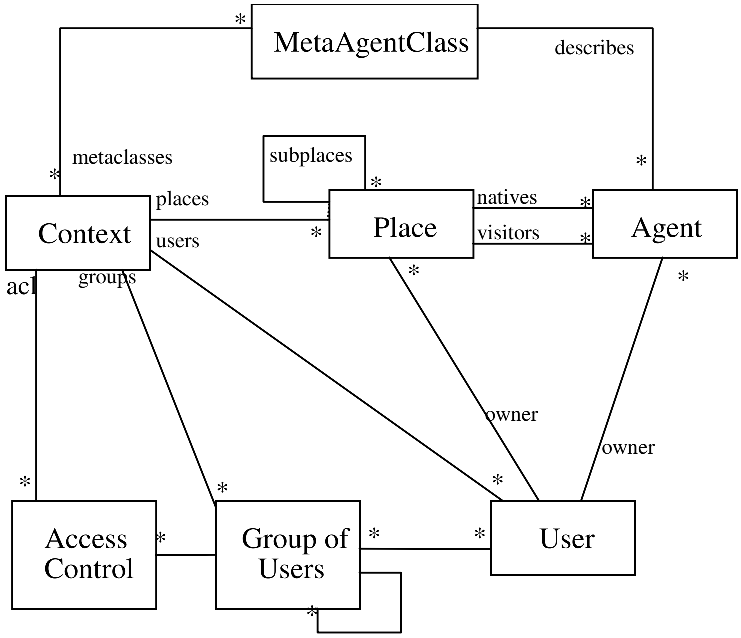

Figure 9: Class diagram of AgentSpace's main concepts.

The AS-Server also maintains lists of *users, groups of users* and *acl* to implement the permission and access control mechanism. A user can belong to one or more groups. Groups can be hierarchically organized to simplify permission management. This means that all users of some specialized group have implicitly all the permissions they inherit from the more general groups. By default, every AS-Server defines four groups of users and establishes a convenient security access policy, based on them: anonymous group; end-users group; place owners group; and AS-Server's administrators group.

7.3 The Agent Pattern in AgentSpace

Figure 10 shows the structure of the Agent pattern specific to the AgentSpace framework. Due to the fact that AgentSpace has been developed on top of the Voyager infrastructure, the persistence and distributed details of the Agent pattern are transparently supported by an internal class (i.e., not visible from the programmer's perspective) which is called `InternalAgent`. (It is out of the scope of this paper to describe the design of the internal structure of this and related classes.)

It is important to note how suitable, to support dynamic and distributed applications, the process of creating agents (as well as places) can be. Firstly, there is no use or explicit reference to network-enable classes. Secondly, all agents are created through a factory method (i.e., the `createAgent method`) in a transparent, clean and easy way.

Thirdly, AgentSpace provides a very extensible and elegant way to handle security policies/strategies related to the access and interactions between agents and end-users, and between agents themselves. Basically, one security policy/strategy (i.e., a security manager class) is attached to the agent object at its creation. In AgentSpace the `SecurityManager` is an abstract class from which other classes should be derived. By default, every agent is attatched to the `DefaultAgentSM` class. However, other classes – other agent's security policies – can be defined and used by the system through the class loading and reflection mechanisms of Java.

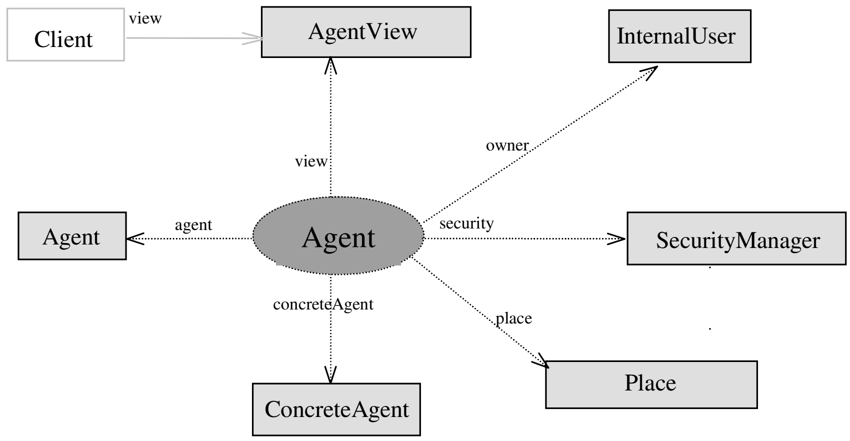

Figure 10: Specific structure of the AgentSpace's Agent pattern – collaboration diagram.

Another novel aspect of AgentSpace is the well-integrated association between users and agents/places. This mechanism, intrinsic by default in AgentSpace, provides an easy and clean way to develop and manage this class of applications.

7.4 The Agent Pattern in Aglets WorkBench

Figure 11 shows the structure of the Agent pattern adapted to the Aglets Workbench [IBM97]. There are some minor differences from the proposed pattern related to some name particularities. (e.g., `Aglet` instead of `Agent`, or `AgletProxy` instead of `AgentView`). In Aglets, an agent is executed in some computational environment which is addressed through a `Context` reference.

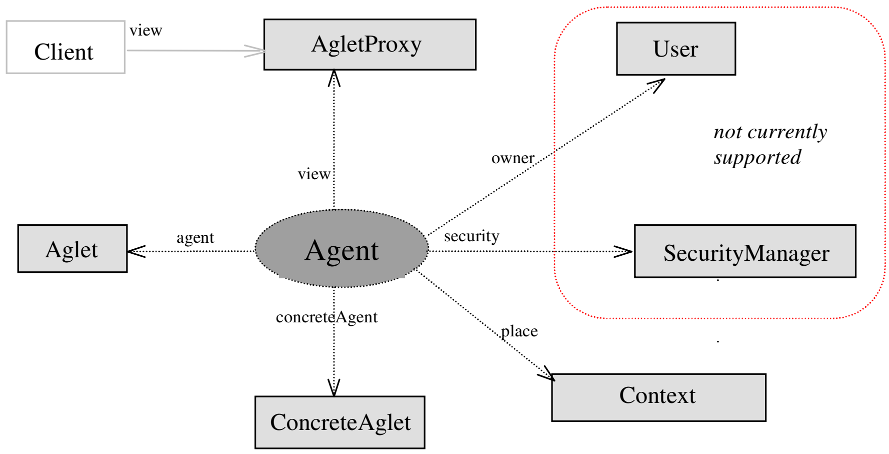

Figure 11: Specific structure of the Aglets' Agent pattern – collaboration diagram.

Finally, it is important to note the inexistence of the `User` and `SecurityManager` references from the Aglets' agent. Maybe in the future IBM will introduce, in a integrated way, this information.

7.5 The Agent Pattern in Telescript

Figure 12 shows the structure of the Agent pattern adapted to the Telescript system [Whi94, Whi96]. There are some semantic differences from the proposed pattern, but nevertheless the similarities are evident. One important difference concerns the way agents are accessed and managed in Telescript: client objects usually access agents directly (when they meet in the same place) or through connections defined at run-time (e.g., based on socket objects). An other basic difference between Telescript and AgentSpace concerns the notion of place. In Telescript a place is an active object with a well-defined behavior and interface (like a stationary agent). On the other hand, in AgentSpace a place is just a static object providing some capabilities and an interface to the computational environment (i.e., the AgentSpace's context).

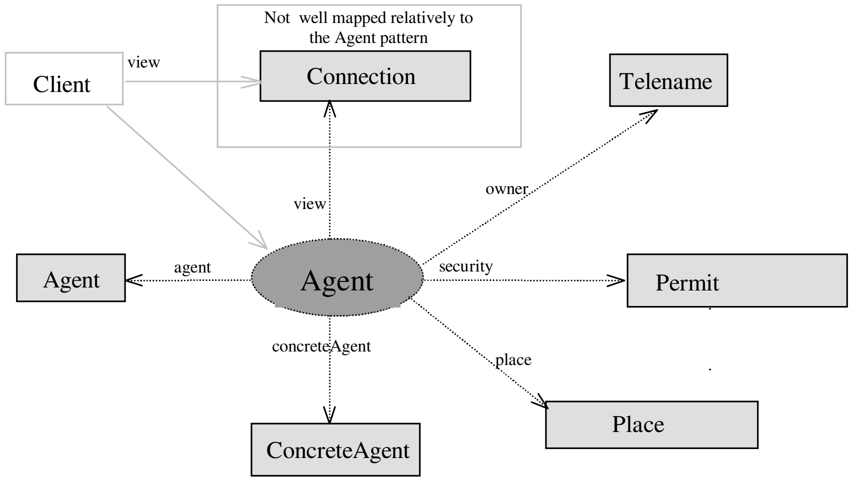

Figure 12: Specific structure of the Telescript' Agent pattern – collaboration diagram.

8 Sample Code

Due to space limitations we present the sample code based on Java classes from the AgentSpace framework. We just present some of the visible classes involved in the Agent pattern, namely the `Agent`, `AgentView` and `ConcreteAgent` types.

The `Agent` is, as mentioned above, an abstract class.

```
public abstract class Agent Cloneable, Serializable
{
transient private InternalAgent iagent= null;

protected Agent() {}

/**
 * public final methods
 */
public final InternalUser getOwner() { return this.iagent.getOwner();}
public final Object clone() throws CloneNotSupportedException {
   return this.iagent.clone();.
}

public final void moveTo(PlaceId p, Ticket t, String cb) throws AgentSpaceException {
   this.iagent.moveTo(pid, t, callback);
}

public final void moveTo(PlaceId pid, Ticket t) throws AgentSpaceException {
   this.iagent.moveTo(pid, t);
}

...

/**
 *  callback methods
 */
public void onCreation(Object init) { }
public void run() {}
public void beforeMove() throws AgentAbortException {}
...

}
```

The `AgentView` is just an interface which is implemented by the `AgentViewImpl` class (which is not visible from the programmer perspective).

```
public interface AgentView
{
public String getClassName() throws AgentSpaceException;
public Class getAgentClass() throws AgentSpaceException;
public InternalUser getOwner() throws AgentSpaceException;
public Object clone() throws /*AgentAbortException,*/ CloneNotSupportedException;
public void moveTo(PlaceId pid, Ticket t) throws AgentSpaceException;
public void moveTo(PlaceId pid, Ticket t, String callback) throws AgentSpaceException ;
...
}
```

The `ConcreteAgent` is just a subclass of the `Agent` class. Let us consider in this example two agent classes interacting through the master-slave model. For the sake of readability, we have purposely removed the exception handling code. This example shows the clean and easy process to create agents, and how they interact through `AgentView` objects.

The `MasterAgent` instance creates dynamically another agent, which is an instance of the `SlaveAgent` class. The operation `createAgent` creates a new agent (instance) with the default security policy, and with the same user as its creator. The AgentSpace handles all these associations implicitly and transparently.

The `MasterAgent` code:

```
public class MasterAgent extends inesc.as.agentspace.Agent
{

 public void run()  {
   PlaceView place= getCurrentPlace();
   // create slave agent
   AgentView av= place.createAgent(this.getOwner(), "SlaveAgent");
   av.start();
   // sleep for a while
   try{ Thread.sleep(4000); } catch (InterruptedException e) {}
   // sleep for a while
   av.sendMessage(new Message(getId().toString(), "doSomething"));
 }

 public void handleMessage(Message m) {
   if ("dienow".equalsIgnoreCase(m.getKey())) {
     System.out.println("MasterAgent: i'll keep alive :-)");
   }
   else
     System.out.println("MasterAgent: ecco " + m.getKey());
 }
 ...
}
```

The `SlaveAgent` code:

```
public class SlaveAgent extends inesc.as.agentspace.Agent
{
 public void run()  {
   System.out.println("SlaveAgent: i'm alive");
 }

 public void handleMessage(Message m) {
   if ("doSomething ".equalsIgnoreCase(m.getKey())) {
      System.out.println("SlaveAgent: receive ‘doSomething’ msg");
      ContextView cv= getCurrentContext();
      AgentView av= cv.getAgentOf(m.getSender());
      av.sendMessage(new Message(getId().toString(), "reply-doSomething"));
   }
   else
     System.out.println("SlaveAgent: ecco " + m.getKey());
 }
 ...
}
```

9 Known Uses

The pattern described in this paper is as seen above used in AgentSpace, Aglets and Telescript frameworks. We expect that it will be probably used, eventually with minor differences, in the emerging mobile agent systems, mainly those based on the Java virtual machine.

Aglets Workbench, for instance, has a simplified implementation of this pattern. It doesn't provide the dynamic and flexible association between agent classes, users and agents' security managers provided by AgentSpace. Nevertheless, it provides the `AgentView` functionalities, namely through the `AgletProxy` instance. On the other hand, Telescript's connections and permits have not the same behavior and semantic than the equivalent `AgentView` and `SecurityManager` proposed in the Agent pattern.

10 Related Patterns and Frameworks

The Agent pattern uses some adapted versions of well-known design patterns. Namely:

- The `AgentView` class presents some similarities with the *Proxy* [GHJV95] and *Remote Proxy* [BM+96] patterns.
- The `SecurityManager` class and its subclass hierarchy (as was developed in AgentSpace) is inspired in the *Strategy* [GHJV95] pattern.
- The `ConcreteAgent` dynamic creation is based on the *Factory Method* [GHJV95] pattern.
- The `Agent` class istself may be viewed as an adaptation of the *Active Object* pattern [LS95].

The right design/application of these different patterns is very important to support the dynamicity of the agent-based applications as referred to above.

Kendall et al. developed one interesting and preliminary work in agent patterns [KM96, Ken+97]. They discussed and proposed a set of well-known patterns (such as *Active Object*, *Mediator*, *Proxy*, *Adapter*, *Negotiator*, and so on) used to support basically the „strong agent" vision [WJ95], i.e., the agent viewed by the artificial intelligent community [DM90, Rie94].

Figure 13 shows the relationships between the different classes of frameworks involved with the support and development of dynamic and distributed agent-based applications.

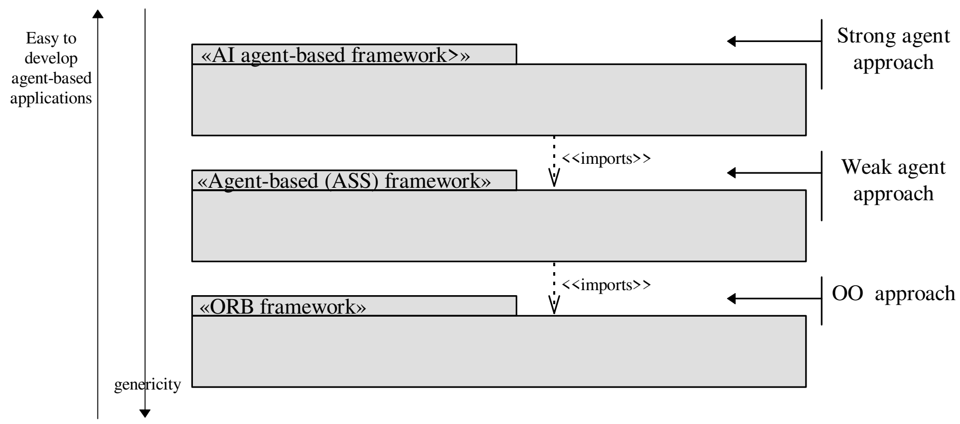

Figure 13: Interrelated frameworks to support agent-based applications.

Basically we identify three interrelated classes of frameworks each one using/importing features provided by the previous one.

ORB frameworks should exist at the bottom level. They provide very powerful but generic features based on the object-oriented approach, such as persistence, name service, communication, location transparency, mobility, etc. Examples of this class of frameworks are Iona's OrbixWeb [Ion98], Inprise's VisiBroker [Inp98], Sun's RMI [Sun97], and ObjectSpace's Voyager [Obj97].

At the middle level and based on the services provided by ORBs, agent-based frameworks can be defined. These frameworks are less general than the equivalent ORBs, and should provide a well-integrated set of components in order to facilitate the development of agent-based applications (basically following the „weak agent“ vision [WJ95]). The main issues handled by these frameworks are those referred to in Section 7.1. Examples of this class of frameworks are those mentioned in the Section 7 of this paper (i.e., AgentSpace, Aglets and Telescript) and others such as AgentTcl [Gra95], Odyssey [GM97], ffMain [LDD95], Grasshopper [IKV98].

Lastly, on top of the previous frameworks there should exist (what we call by) „agent-based applicational“ frameworks. These kinds of framework should provide yet more specific application components such as more agent specializations. One possible approach to these frameworks may be artificial intelligent frameworks – based on knowledge representation, high-level communication protocols/languages, agent-based models (e.g., the BDI model [RG95]), etc. RMIT [Ken+97] and Plangent [Ohs+97] are examples of these last kind of (applicational) frameworks.

The agent pattern proposed in this paper is mainly focused on architectural aspects related to agent-based frameworks (the middle level) referred above. It is our conviction that on the top of this low-level and application independent agent pattern, more work should be developed, in particular several patterns described in Kendall work [Ken+97] such as the *Adapter*, *Negotiator*, *Mediator* and *Reasoner* patterns.

Acknowledgments

We would like to thank our shepherd Doug Lea to help improve the pattern mainly in content and structure.

References

[BM+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons. 1996.

[COS97] Ponton, COGEFO/CEFRIEL, Hamburg University, INESC, Interzone Music Publishing, Oracle UK, and SIA. *COSMOS – Common Open Service Market for SMEs*. ESPRIT Research Project Proposal. 1997. Starting June 1998.

[DM90] Y. Demazeau, J. Muller. Decentralized Artificial Intelligence. *Decentralized Artificial Intelligence*, Y. Demazeau, J. Muller (editores), Elsevier, 1990.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman. 1995.

[GM97] General Magic, Inc. *Odyssey Product Information*. 1997. `http://www.genmagic.com/agents/odyssey.html`

[Gra95] R. Gray. AgentTcl: a Transportable Agent System. *Proceedings of the CIKM Workshop on Intelligent Information Agents*, (CIKM'95), 1995.

[IBM97] IBM Tokyo Research Laboratory. *The Aglets Workbench: Programming Mobile Agents in Java*, 1997.

[IKV98] IKV++ GmbH. *Grasshoper, An Intelligent Mobile Agent Platform written in 100% pure Java,* 1998. `http://www.ikv.de/products/grasshoper/`

[Inp98] Inprise Corp. *VisiBroker – Distributed Object Connectivity Software*. 1998. `http://www.inprise.com/visibroker/`

[Ion98] IONA Technologies. *OrbixWeb 3 – The Internet ORB*. 1998. `http://www.iona.com/products/internet/orbixweb/`

[Ken+97] E. Kendall et al. The Layered Agent Pattern Language. *Proceedings of the Conference on Pattern Languages of Programs* (PLoP'97), 1997.

[KM96] E. Kendall , M. Malkoun. The Layered Agent Patterns. *Proceedings of the Conference on Pattern Languages of Programs (PLoP'96)*, 1996.

[LDD95] A. Lingnau, O. Drobnik, P. Domel. An HTTP-Based Infrastructure for Mobile Agents. *WWW Journal* (Fourth International WWW Conference), W3C, 1995.

[LS95] R. Lavender, D. Schmidt. Active Object: an Object Behavior Pattern for Concurrent *Proceedings of the Conference on Programming. Patterns Languages of Programming*, 1995.

[Obj97] ObjectSpace. ObjectSpace *Voyager Core Package Technical Overview*. 1997.

[Ohs+97] A. Hshuga et al. Plangent: An Approach to Making Mobile Agents Intelligent. *IEEE Internet Computing*, 1(4), 1997.

[Rat97] Rational Software Corp. *UML – Unified Modeling Language*, version 1.0. 1997.

[RG95] A. Rao, P. Georgeff. BDI Agents: from Theory to Practice. *Proceedings of the First International Conference on Multi-Agent Systems*, 1995.

[Rie94] D. Riecken (editor). Special Issue: Intelligent Agents. *Communications of the ACM*, 37(7), Julho 1994.

[RM98] A. Romão, M. Mira da Silva. An Agent-Based Secure Internet Payment System for Mobile Computing. *Proceedings of the International Conference on Electronic Commerce'98*, (Hamburg, Germany) 1998.

[SMD97] A. Silva, M. Mira da Silva, J. Delgado. Motivation and Requirements for the AgentSpace: A Framework for Developing Agent Programming Systems. In *Proceedings of the Fourth International Conference on Intelligence in Services and Networks* (IS&N'97, Cernobbio, Italy) 1997.

[SMD98] A. Silva, M. Mira da Silva, J. Delgado. AgentSpace: An Implementation of a Next-Generation Mobile Agent System. *Mobile Agents'98*, (Stuttgart, Germany) 1998. (To be published.)

[Sun97] Sun Microsystems, Inc., JavaSoft. *Java Remote Method Invocation (RMI)*. 1997. `http:// www.javasoft.com/products/jdk/rmi`

[Whi94] J. White. *Telescript Technology: The Foundation for the Electronic marketplace*. General Magic. 1994.

[Whi96] J. White. General Magic, Inc. *Mobile Agents White Paper*. 1996.

[WJ95] M. Wooldridge, N. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*. 10(2), 115-152. Cambridge University Press, 1995.