An Overview of AgentSpace: A Next-Generation Mobile Agent System

Alberto Silva, Miguel Mira da Silva[1] and José Delgado

{Alberto.Silva, Jose.Delgado}@inesc.pt, [1]mira-da-silva@p.pt
INESC & IST (Technical University of Lisbon)
Rua Alves Redol, nº 9, 1000 LISBOA, PORTUGAL

Abstract. This paper gives an overall overview of the AgentSpace framework, a next-generation Java mobile agent system developed on top of the ObjectSpace Voyager system. We first introduce the notion of dynamic and distributed agent-based applications and argue that the AgentSpace features are suitable to support them. The AgentSpace novelties include: flexible and dynamic association between agents, security policies and users; transparency of agent location through the use of views; and easy and clean way to create agents through the use of abstract classes and method factories. The paper uses an application example to present some aspects of AgentSpace from the developer's point of view.

1 Introduction

Nowadays, there are many Mobile Agent Systems (MAS) proposals – such as Aglets [1], Agent-Tcl [2], Odyssey [3] or Tacoma [4] – that have roughly the same purpose and present a common set of functionalities. Nevertheless, they also present some important technical and even conceptual differences.

For instance, Telescript [5] used to be the usual reference for MAS with a very high technological level. However it is (or it was) a proprietary system, with a difficult-to-learn programming language, and badly suitable to a dynamic and open environment such as the Internet. On the other hand, a system such as ffMAIN [6] is language and system independent. However, it shows severe limitations on the overall performance and difficulties in developing complex applications. The Aglets Workbench – today the reference Java-based MAS – lacks an elaborate object model, e.g., without the notion of execution places hierarchically organized; without management operation on agent families and clusters. It also lacks some technical capabilities, e.g., just two ACLs (access control levels); without the notion of an agent class manager, without the "open channel" capability [18], or even without the notion of users transparently associated to agents.

We expect that, in the future, these MASs will be improved in order to support the development and execution of more flexible, reliable, secure and efficient agent-based applications (ABA). In order to achieve that objective, these MASs should

incorporate a good combination of feature from existing MASs. For example, they should support the independence protocol of TCP/IP and/or HTTP-based MAS, the technology for migrating executing agents (threads) as found in Telescript, and the tight integration with Java as supported by the Aglets Workbench.

In a recent paper [7] we have proposed a conceptual MAS architecture composed by three complementary components: AES, for Agent Execution System; ACS, for Agent Class System; and AEE, for Agent Execution Environment. We argue these components should be the "building blocks" of a next-generation MAS. We have also identified other related components needed to develop, manage and monitor agent-based applications.

Based on that preliminary work, a Java-based framework was implemented. This framework – called *AgentSpace* – was developed on the top of Voyager [11] from ObjectSpace.

In this paper we overview the main aspects of AgentSpace from a developer's point of view, namely its exported API (Java interfaces and classes). AgentSpace will be further improved to be used in the ESPRIT COSMOS project [13].

The paper is organized as follows. In Section 2 we present our own definition of agent and agent-based application, and also introduce the main aspects of Voyager. In Section 3 we overview the AgentSpace architecture and object model. In Section 4 the AgentSpace API is described. In Section 5 we use an application example to present some aspects of the AgentSpace API. Finally, Section 6 summarizes the contributions with a small discussion of agent-based applications that will be consider in the future.

2 Agents and Agent-Based Applications

Due to the proliferation of agent definitions with different point of views, scopes and possible applications – see for example [15, 16, 17] – we start this paper by defining the meaning of an agent in our work context.

2.1 Agents and Mobile Agent Systems

In this paper, an *agent* is a software entity with a well-known identity, state and behavior, with autonomy to somehow represent its user. The agent's user might be a human or an organization (enterprise, community, etc.) as well as another agent.

From a more technical point of view, an agent can be implemented as an active object of medium granularity. This means that an agent is an instance of some defined class – with its own group of threads, state and code – identified by a unique global identity.

From a higher-level, conceptual perspective, the agent is a basic but powerful concept to think about and to design complex, distributed, dynamic applications as those enabled by the Internet.

From yet another perspective – the human-computer interaction perspective – agents may be viewed as a new interface paradigm to help end-users access future Internet applications, including electronic commerce. In this world, 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., information search). Using agents, users can delegate a set of tasks to be done by agents, instead of doing these tasks themselves directly. This new paradigm is especially attractive to help users in complex, tedious or repetitive tasks in open, dynamic, vast and unstructured information sources such as those found on the Internet.

Despite the fact that AgentSpace framework also supports mobile agents, specific agent attributes – such as intelligence and mobility – are not, from our point of view, fundamental to define a software object as an agent. We agree, however, that both intelligence and mobility can be important for developing certain kinds of applications, depending on their requirements. However, it is important to state that not all agents in agent-applications will be intelligent and/or mobile.

On the other hand, intelligence is neither the focus of our research work, nor particularly interesting for this workshop. At least, intelligence characterized as a formal method to define, manage and exchange knowledge in a higher abstraction level, since normal Java code can also represent intelligence [19].

2.2 Agent-based Applications

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 by itself. An ABA is not a typical application that is owned and managed by some people 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.

ABA applications have a number of characteristics and requirements that have been dealt with independently in the past. It is their combination that poses problems.

- *Autonomous*: Each user creates and maintains their own agents using their own resources and/or using resources from others.
- *Heterogeneous*: Each user has bought, got used to and used different interfaces, machine architectures, programming languages, database systems, communication packages, operating systems and so on.
- *Open*: Some agents may depend on other agents and applications, even from external organizations. This means agents will have to inter-operate with other (legacy) information systems (applications, databases and so on).
- *Dynamic*: Agents will be added, updated and removed at any time without previous notice. They will have to cope with unavailability, new interfaces, oscillating bandwidths, and other variable characteristics.

- *Robust*: Agents will have to tolerate different kinds of failures on machines, networks or at any level of software. For example, agents cannot stop executing just because a company is rebooting their gateway to the outside world.
- *Secure*: The system should provide different levels of security depending on each particular part of the whole application. There will be public, place-specific and administrative access control lists.

All these characteristics make ABA potentially very difficult to implement and use. However, we believe MAS, and in particular AgentSpace, will help developers build and manage them.

2.3 An Overview of Voyager

Due to the fact that Voyager was used to support the development of AgentSpace, we overview it briefly in this section.

ObjectSpace's Voyager [11] is a commercial product to support the development of distributed Java applications. Amongst other features, Voyager provides a "100% Java" ORB (remote method invocation) well designed and integrated with the JDK 1.1 class interfaces. Voyager can be seen as a traditional CORBA object request broker (ORB) with extended features such as object mobility and life spans.

As a communication infrastructure, Voyager provides capabilities to: (1) create objects that can be called remotely; (2) obtain references to remote objects; and (3) send messages to, and call methods on, remote objects. Of course, all these operations also apply to local objects. As an ORB, Voyager extends these operations to remote objects in a transparent way. This means the program – and thus the programmer – can treat both local and remote objects in the same manner without distinguishing between these two types of objects.

In addition, Voyager provides many other functionalities, such as: (1) persistency – objects may outlive the program that created them; (2) distributed garbage collection – an extension to the local Java garbage collection; (3) a number of messaging modes; (4) naming service; (5) object mobility – so that objects can be exchanged between Java programs; (6) distributed events; (7) publish/subscribe mechanism; (8) group communication; and (9) interoperation with CORBA objects.

ObjectSpace claims that Voyager is also a *mobile agent system*, going to the point of comparing it with Odyssey, Aglets and Concordia [12]. However, in our opinion, Voyager and these truly mobile agent systems are not comparable because Voyager only supports mobile objects without any of the agent support typical in other agent systems. Instead, Voyager should be better comparable to Sun's RMI for which it represents a much better alternative – more features and still faster and simpler to learn and use.

Despite Voyager's powerful capabilities and overall elegance, *Voyager offers only limited support* to the kind of applications described in Section 2.2 above. These applications have requirements regarding openness, dynamicity, and access control mechanisms that are (at least currently) lacking in Voyager. Voyager's "agents" are not truly software agents as we defined in Section 2.1 above because they are just

objects with mobility support and a given life-span (one day by default). For example, these objects don't keep any specific information regarding their owner or their native and current execution place.

On the other hand, Voyager-based applications require the creation and management of so-called "virtual classes" for all potentially remote and/or mobile objects. A virtual class is an RPC-like stub that is generated at compile-time with the ObjectSpace's `vcc` tool from any regular Java interface or class, in either its source or byte-code format. This requirement makes impractical – thus difficult – the development of complex (with a large number of remote and mobile objects used by many different programs) and dynamic (with apparently static objects that suddenly decide to move) applications.

The conclusion is that a higher-level agent-based framework should be designed and built in order to develop the new kind of agent-based applications. In this paper we propose AgentSpace as such a framework that is being built on the top of Voyager. As a consequence, AgentSpace is able not only to provide the majority of Voyager's features but also new ones to properly support software agents.

3 An Introduction to AgentSpace

3.1 Architecture

AgentSpace's main goals are the support, development and management of ABAs as described in Section 2. These goals are provided through three separated but well-integrated components as depicted in Fig. 1.

Both server and client components run on top of Voyagerand 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.

The *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 (UID); (7) support for agent communication; and (8) optionally a simple interface to manage/monitor itself.

The *AgentSpace client* (*AS-Client*) supports – depending on the corresponding user access level – the management and monitoring of agents and related resources. The AS-Client is a set of Java applets stored on an 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. Furthermore, the AS-Client should be able to access several AS-Servers, providing a convenient trade-off between integration and independence between these two components.

The *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 in the AS-Server's database) in order to provide an interface to agents.

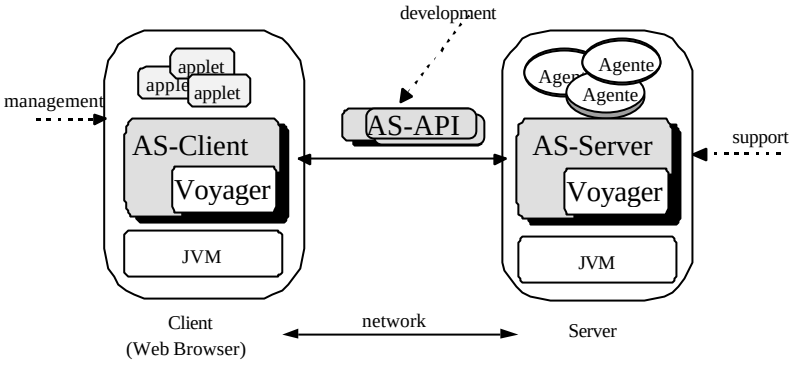

Fig. 1. AgentSpace architecture

These clients/applets can be either generic mini-applications – such as the AS-Client itself, see above – or specific to some particular agent, for example, to input data or present a report.

3.2 Object Model

AgentSpace involves the support, development and management of several related objects: contexts, places, agents, users, groups of users, permissions, ACLs (access control lists), security managers, tickets, messages, and identities. Fig. 2 shows the relationships between these objects through an UML [21] class diagram.

The *context* is the most important and critical object of the AS-Server, as each AS-Server is represented by one context. 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 metaphor 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 place has a unique global identity and knows the identification of its owner/manager. It also maintains a keyword/value list that allows an informal characterization. Optionally, places can be hierarchically organized. The place can also contain the maximum and current number of agents allowed in order to support some resource management. In order to keep track of its agents, the place keeps a list containing its visitant agents and another with its native agents. The place also knows in which place its native agents are executing at a given point of time.

The *agent* is the basic element of the system. Agents are identified by a unique global identity. Agents have two parts: (1) a visible component, that should be developed, or specialized, by programmers (more on this later on); and (2) an invisible component, called "internal-agent", kept by AgentSpace. 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.

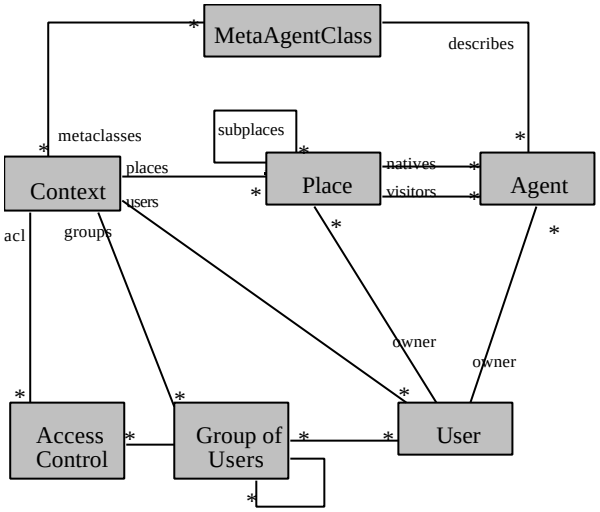

Fig. 2. UML class diagram of AgentSpace

Just one user owns an agent. Nevertheless, other users (or even agents from other users) might interact with it, if this is granted by the agent' security policy.

The AS-Server also maintains lists of *users, groups of users* and *acl* to implement the permission and access control mechanism. A user may belong to one or more groups. Groups may 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 establish a convenient security access policy, based on them: anonymous group; end-users group; place owners group; and AS-Server's administrators group.

4 AgentSpace API

In the previous section we introduced the main concepts related to the AgentSpace design. Nevertheless, the objects that implements these concepts cannot be used directly by ABA programmers. This is not possible in order to provide a flexible security policy and to hide several complexities from the developer (such as those related to distribution, persistence, mobility, and so on).

In this section we concentrate on the main aspects of the interface offered by AgentSpace. There are two complementary uses of this API: to develop agents and AgentSpace-based applets; and to develop generic client tools, such as the AS-Client as mentioned in Section 3.1.

These interface elements (Java objects, classes and interfaces) are organized together as the `inesc.as.agentspace` package, that basically defines the public AS-API.

4.1 Identities

Each element of the AS-Server that is potentially accessible has a unique global identity, namely the `ASId, PlaceId` and `AgentId` types identify respectively an AS-Server's context, a place and an agent.

- `ASId` – a DNS or IP machine address and a server contact port (e.g., "*xico.inesc.pt")*.
- `PlaceId` – an ASId identity; and a unique name based on a sequential counter (e.g., "*xico.inesc.pt:777/pid001"*).
- `AgentId` – an PlaceId identity where the agent was created; and a unique name based on a sequential counter (e.g., *"xico.inesc.pt:888/pid001|aid002"*).

4.2 Users

The user is identified by a unique identity – represented by the `User` class – which contains: user's name; a public key; a set of certificates; the organization and country the user belongs to; and user's e-mail. In spite of all these attributes, the name is the single mandatory field. `Identity` and `Principal` types belong to the `java.security` Sun's package.

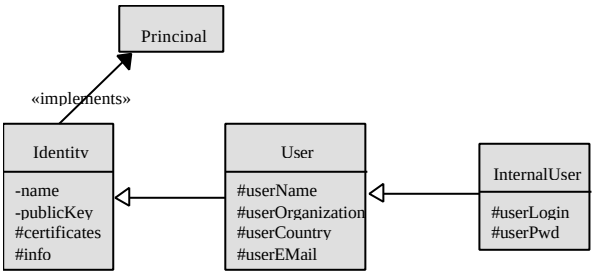

Fig. 3. User classes handled by AgentSpace

Moreover, the user may have different identifiers depending on the context the user belongs to. This specific identity is represented by the `InternalUser` class as depicted in Fig. 3, which contains, in addition to all referred fields authentication attributes (e.g., login and password).

4.3 Views and Security Policies

Together with the factory methods (see Section 4.4 below), views are an important design pattern to support the dynamicity of the agent-based application as referred in Section 2.2.

We have adapted the *Proxy* design pattern [20] at different levels of the AgentSpace design, namely, at the context, place and agent level. This design pattern, which we call "*View*", is very suitable to support transparent and secure access to these different types of objects.

- `ContextView` provides controlled access to the AS-Server's context. Depending on the user's authentication, the `ContextView` object enables, or not, a set of general operations. Namely, operations to manage users, groups of users, permissions, and execution places. Examples of these operations are `createPlace`, `groups`, `createUser`, `removePlace`, `getPlaceOf`, `etc`.
- `PlaceView` provides a controlled access interface to a specific place. Examples of operations protected through place views are: `createAgent`, `removeAgent`, `save`, `flush`, etc. For example, an user can create an agent in some place only if that operation is allowed by that place's security manager.
- `AgentView` provides access to agents independently of their current place. This access goes indirectly through views in order to protect agents and to hide their current localization. Additionally, the `AgentView` class avoids the need to create and manage network-based classes (e.g., virtual objects in the Voyager system). Examples of operations protected through agent views are `getCurrentPlace`, `sendMessage`, `getClassName`, `start`,and `moveTo`.

AgentSpace provides three distinct levels to define security policies, namely at the context, place, and agent level. These different policies are defined and managed dinamically and in a independent way amongs themselves. For example, a security policy defined to some context might be liberal, while the security policy to some place, in this same context, might be extremely restrictive.

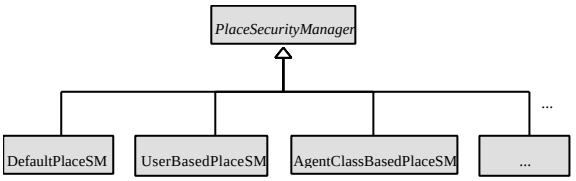

Fig. 4. Possible hierarchy of place's security managers

The actual policy at the *context level* is based on the existence of four groups of users associated to four distinct profiles, namely: administrators; execution places managers; recorded users; and anonymous users. The access to the context is established through some `ContextView` instance, based on which the execution of some operation is, or is not, allowed.

On the other hand, the security policies at the *place and agent levels* are defined through the dynamic association of specific classes. These classes (respectively,

specialization of the `PlaceSecurityManager` and `AgentSecurityManager` abstract classes) may be developed by ABA's programmers in an easy way and are associated to places and agents at their creation time. In that way, it is possible the definition of different security managers, for example, some of them based on user information, other based on agent classes information, etc. Fig. 4 presents a class diagram depicting a possible hierarchy related to place's security managers.

4.4 Factories

AgentSpace provides design patterns to allow the dynamic creation of the following objects: instances of agents and places, and references (views) to agents, places, and contexts. Namely, the *abstract factories* and *method factories* design patterns [20].

Table 1 summarizes the main classes of the AS-API involved with the creation and generation of references to the main involved objects. It is particularly important to note that the access and manipulation of these specific methods (e.g., `createAgent` and `createPlace`) is controlled by their respective views.

Table 1. Factories Methods in AgentSpace

Purpose	Factory method		Result object
	Base Class	Method	
Context:			
- reference generation	*AgentSpace* *AgentView*	*getContextView* *getCurrentContext*	*ContextView*
Execution Place:			
- creation	*ContextView*	*createPlace*	*PlaceView*
- reference generation	*ContextView* *AgentView*	*getPlaceOf, places,* *myPlaces* *getCurrentPlace*	
Agents:			
- creation	*PlaceView* *Agent*	*createAgent* *clone*	*AgentView*
- reference generation	*ContextView* *PlaceView*	*getAgentOf* *getNatives, getVisitors,* *queryAgents*	

In Section 5 we will show how to use some of the methods above, namely how to create an agent, and how to obtain context, place and agent view references.

4.5 Agents

Agents are the extensible elements of AgentSpace. Basically, programmers should derive the `Agent` abstract class in order to build their own agents. The `Agent` class has three main groups of methods: (1) final; (2) callbacks; and (3) helper methods, as depicted in Fig. 5. (The `Agent` subclass is represented by the `ConcreteAgent` class.)

- *Final methods* are pre-defined operations provided to all agents 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 to be 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 another agent, an end-user (via same applet), a time service, etc. The callback mechanism provides the desired extensibility of the agent component. Examples of callbacks are: `run`, `onCreation`, `beforeDie` and `handleMessage`.
- Finally, agent classes also have *helper methods* (usually with private or protected access modifiers) in order to support specific functions of that class/object.

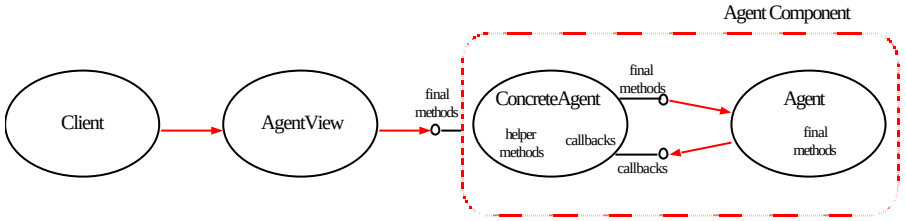

Fig. 5. Agent's main groups of methods in AgentSpace

Helper methods should be defined in concrete agent classes and are used internally by callback methods. On the other hand, callbacks should be specialized in concrete agent classes, and final methods can only be used/invoked either directly by concrete agent classes or indirectly through `AgentView` interfaces. Additionally, some callbacks may be defined in order to be called after the navigation operations. These methods are referred explicitly as a parameter in `moveTo` methods.

Table 2. Events and methods associated to the agent execution

Operation/Event	Final Method	Correspond Callback
Life cycle:		
- Creation	*createAgent*(3)	*onCreation*
- Activation	*start*(1)	*run*
- Clone	*clone*	*beforeCloning; run*
- Memory Management	*flush*	*beforeFlush; afterLoaded*
- Persistence	*save*	-
- Delete	*die*	*beforeDie*
End-user interaction:		
	getUserInicialization(3)	*doUserInicialization*
	getUserConfigurationn(3)	*doUserConfigurationn*
Navigation:		
	moveTo	*run; or atPlaceX*(2)
	backHome	*afterBackHome*
Event Notification:		
	AddAgentListener(1)	-
	sendEvent	
Communication:		
- Asynchronous (by default)	*sendMessage*(1)	*handleMessage*

- Synchronous (by default)	*doOperation*(1)	*handleOperation*

(1) These methods are invoked only through the `AgentView` interface

(2) Callback not predefined – defined by the programmer at concrete classes

The final methods of the `Agent` class can be analyzed following two main groups: management methods; and get/set methods.

The first group concerns on methods associated with life cycle, navigation, user interaction, and communication operations. The execution of these methods involve, in general, the invocation of a correspondent callback as summarized in Table 2.

Final methods defined in the `Agent` class can be invoked directly in their subclasses without the utilization of any `AgentView` instance. This means that in some concrete agent class it is possible to specify autonomous behaviors, for example, an agent may decide to move itself to another place (`moveTo`) or to its native place (`backHome`), or suspend itself (`flush`), etc.

Table 3. Get and Set final methods of the `Agent` class

Purpose	Final Method	Result Type
Get/Set the agent's informal description	*getDescription*	*String*
	setDescription	-
Get agent's identifier	*getId*	*AgentId*
Get the agent's native place identifier	*getNativePlaceId*	*PlaceId*
Get the agent's current place identifier	*getCurrentPlaceId*	*PlaceId*
Get a reference to the agent's current place	*getCurrentPlace*	*PlaceView*
Get a reference to the agent's current context	*getCurrentContext*	*ContextView*
Get the agent's class name	*getClassName*	*String*
Get the agent's class	*getAgentClass*	*Class*
Get the agent's meta agent class	*getMetaAgentClass*	*MetaAgentClass*
Get the agent's owner	*getOwner*	*InternalUser*
Get/Set the agent's properties	*getProperty*	*String*
	getProperties	*Enumeration*
	setProperty	-

The second group of final methods doesn't imply the execution of any type of callback. These methods (see Table 3) just allow the getting and setting of information associated to each specific agent.

The `getMetaAgentClass` method returns a particular type – the `MetaAgentClass` class –, which provides a flexible mechanism to handle agent classes existent in every AS-Server's context.

4.6 Other Classes

The AS-API also provides other related types that will not be further described in this paper:

- `Message` – to provide inter-agent communication;
- `Ticket` – to provide agent mobility with an access control mechanism;
- `AgentSpace` – to provide a context view factory;

- The `inesc.as.security.acl` package that includes classes such as: `GroupImpl`, `PermissionImpl`, `AclEntryImpl`, `AclImpl`, and so on – to provide access control management.

It is not the goal of this paper to describe fully all the details of the AS-API. However, in the following section, we will use an example to describe better some interesting aspects, namely: (1) how to create agents; (2) how to obtain references to them; (3) how do they navigate between places; and (4) how do they communicate.

5 Example Application

In this section, we take the perspective of programmers writing agent-based applications and discuss the main aspects related with specializing the `Agent` class.

We will use, as an example, a "VirtualShop" application. The VirtualShop is composed by an open and dynamic set of shoppers, who want to advertise and sell their specific products. A well-known trusted company (such as a Telecom provider or an ISP) manages VirtualShop. For the sake of this example and for simplicity, let's assume that a fictions company (PPT) provides a well-known broker-agent providing several services (such as yellow pages, payment support, audit trail, and so on).

5.1 How to create agents

Agents are instances of any class derived form `Agent`. Agents can be created: either interactively through the AS-Client (or a similar tool), or by any other agent.

Agent creation using AS-Client tools. In the first case, the AS-Client should perform basically the following algorithm, for example, from an applet running in the browse:

```
InternalUser user= AgentSpace.getUserByLogin(asid, "user-
login", "user-pwd");
ContextView cv= AgentSpace.getContextView(asid, user);
PlaceView pv= cv.getPlaceOf("VirtualShop");
AgentView av= pv.createAgent(user,
"ppt.virtualshop.Shopper");
…
```

Firstly, the application needs to get the `ContextView` and the `InternalUser` objects, respectively `cv` and `user`. The user has to specify login information, so that the `cv` object can check permissions.

Then, a reference is needed (typically a remote reference) to the place where the agent should be created. Eventually an exception may be raised, in case the place doesn't exist, or the user cannot access that place. The security strategy may vary between places. For instance, one place may adopt a security strategy based on users' ACL, while another may adopt a security strategy based on a previous agent classes record.

Finally, the agent is created by specifying user information and the agent class name. For security reasons, it is not allowed the agent creation from remote agent classes.

Agent creation from another agent. Let's suppose that the `ppt.virtualshop.Broker` agent class creates an instance of `inesc.virtualshop.Shopper` class in the same place as it is currently running and additionally creates a clone of itself.

The code below shows the two ways to create agents: by explicit agent class specification, or by cloning. Note that the created agents are attached to the current place and have the same owner (`InternalUser`) as the corresponding agent creator. Still, the same security issues should be posed as referred to above.

```
class Broker extends Agent {
  …
  void run() {
    …
    PlaceView pv= getCurrentPlace();
    AgentView        av1=        pv.createAgent(getOwner(),
“inesc.virtualshop.Shopper”);
    av1.start();
    …
    AgentView av2= clone();
  }

}
```

There are other variants to the createAgent method as shown below:

```
public  AgentView  createAgent(InternalUser  user,  String
className)
public  AgentView  createAgent(InternalUser  user,  String
className, Object init)
public  AgentView  createAgent(InternalUser  user,  String
className, Object init,
                            String ssClassName)
```

In all these variants, it is required the specification of the user (`user`) that will be attached as the agent' s owner, as well as the involved agent class (`className`). Optionally, an initialization object (`init`) might also be specified that should be manipulated by the programmer in the `onCreation` callback. Additionally, and also optional, an agent security manager class (`ssClassName`) might be attached to that agent. In case that last variant is not used, the created agent should present a security policy by default represented by the `DefaultAgentSM` class, which is provided by the AgentSpace infrastructure.

5.2 How to obtain references to agents

An `AgentView` object, as seen above, is an agent reference. There are several ways to get an agent view: (1) as the result of agent creation methods (`createAgent` and `clone`, as seen above); as well as (2) through an AgentView factory method.

To obtain a reference to any agent, through an AgentView factory, it is only need to know its corresponding identity (`aid`). With this `aid` the factory method `getAgentOf` can be invoked from the `ContextView` object.

```
AgentId                    aid1=                    new
AgentId("cupido.inesc.pt:777/VirtualShopper|BertrandShop");
AgentView av1= cv.getAgentOf(aid1.toString());
AgentView                                          av2=
cv.getAgentOf("cupido.inesc.pt:777/pid003|aid034");
…
```

5.3 How do agents navigate

In general, Java agents are not able, like other MASs (e.g., Telescript or Agent-Tcl), to keep their execution state after a navigation operation. This limitation is due to the fact that it is not possible (in the current Java version) to access a thread's execution stack.

AgentSpace uses Java's reflection capabilities to give programmers the possibility to specify the callback they want to be invoked after that operation, instead of always calling the same callback after the move/dispatch operation (as the `run` callback in Aglets system),

This approach reduces significantly the current Java limitation, offering a more elegant and simple way to program agent classes as it (avoiding the "spaghetti code" of long switch instructions found for example in Aglets agents).

Before a move operation, every agent needs to have a `Ticket` object in order to be accepted in the target place. A ticket is a certificate object that keeps the information required by the target place security policy in order the agent may be accepted.

There are two final methods concerning agent mobility: `moveTo`; and `backHome`. The second method gives the agent the possibility to go back home, and after that, in its native place, to have its `afterBackHome` callback invoked.

```
class Shopper extends Agent {
  …
void run() {
    …
    Ticket tck= new Ticket(this);
    PlaceId                              pid=              new
PlaceId(“xico.inesc.pt:777/VirtualShopper/BrokerPlace”);
    moveTo(pid, tck, “atBroker”)

  }
  void atBroker () {
    changeMyInfo();
    backHome();
  }

  void afterBackHome () {
    out(“I’m home again!”);
  }
  …
}
```

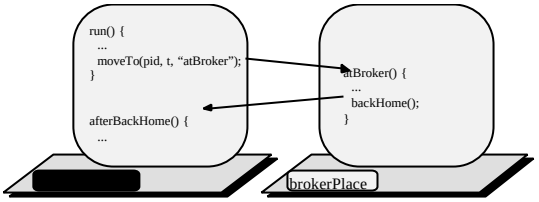

Fig. 6. Agent mobility programming

5.4 How do agents communicate

Agents communicate between themselves in a indirect way through their respective references (i.e., `AgentView` instances). Some final methods can be invoked if the owner of some reference has the required permission. Nevertheless, the majority of the functionalities defined in the concrete agents are not accessible in this way. Consequently, AgentSpace provides two complementary and extensible mechanisms to agent interaction, both of them based on the message exchange: one for unidirectional communication (`sendMessage` method); and other for bi-directional communication (`doOperation` method). Additionally, the programmer can specify the communication semantic it is need, namely: synchronous, asynchronous, and future based.

6 Conclusions

In this paper we presented the AgentSpace mobile agent system as a new Java framework to support, manage and develop future agent-based applications. The focus of this paper was to present an overview to the main components of AgentSpace, its exported object model and programmer’s API.

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

Additionally we provide a very extensible and elegant way to handle security policies related to the access and interactions between agent and end-users, and

between agents themselves. Basically, one security manager is attached to the agent (or place) object just in the moment of its creation.

Other novel aspect of AgentSpace is the well integrated association between users and agents/places. This mechanism, intrinsic by default in AgentSpace, provides a simpler way to develop and manage this class of applications.

One important reason of the AgentSpace's abilities is due necessarily to Voyager, with its very well suitable features, namely support for object mobility and persistence. Due to Voyager, AgentSpace agents may be accessed from applets running on Web browsers, or easily support the communication in the open-channel situation (i.e., agents can communicate transparently even if one has been moved to another place).

Another important reason is obviously due to Java itself with its recent features (JDK 1.1), namely reflection, dynamic class loaders and object serialization. However, we still need more features from the Java Virtual Machine, such as possibility to access current thread execution stack or an even more flexible way to handle security managers.

AgentSpace, version 1.0.2, is ready to be used by other programmers. Amongst other applications, AgentSpace is being used to implement a generic "Agent Manager Tool" (with a graphical user interface, the AS-Client) and a prototypical electronic payment system based on SET [14]. In the future, we will continue developing the AgentSpace in the COSMOS project [13] that will build a framework to negotiate contracts on the Internet. Other experiments are planned as well, such as dynamic updating of components. We hope these applications will provide many criticisms and feedback to further improve AgentSpace.

For the interested reader, we have some recent papers that describe complementary aspects of AgentSpace. In [9] we proposed the *Agent* pattern design, which is suitable to support dynamic and distributed applications. In [10] we describe an electronic-commerce ABA which was developed both on top of AgentSpace and the Aglets Workbench. Based on these prototypes we present some preliminary performance figures comparing AgentSpace with Aglets. We also discuss the advantages of AgentSpace from the programmer's point of view. For a detailed and complete discussion of these (and other) aspects the reader is referred to [8].

We have also put some source code and related information available for download at the following address: http://berlin.inesc.pt/~agentspc/.

References

1. IBM Tokyo Research Laboratory. The Aglets Workbench: Programming Mobile Agents in Java (1997).
   `http://www.ibm.co.jp/trl/aglets`
2. Gray, R.: Agent Tcl: A transportable agent system. In Proceedings of the CIKM'95 Workshop On Intelligent Information Agents (1995).
3. General Magic, Inc. Odyssey Product Information (1996).
   `http://www.genmagic.com/agents/odyssey.html`

4. Hohansen, D., Renesse, R., Schneider, F.: An Introduction to the TACOMA Distributed System. Computer Science Technical Report 95-23. University of Tromso, Norway (1995).
5. White, J.: General Magic, Inc. Mobile Agents White Paper (1994) `http://www.genmagic.com/agents/Whitepaper/whitepaper.html`
6. Lingnau, A., Drobnik, O., Domel, P.: An HTTP-Based Infrastructure for Mobile Agents. In Proceedings of the Fourth Int'l WWW Conference (1995).
7. Rodrigues da Silva, A., Mira da Silva, M., Delgado, J.: Improving Current Agent Support Systems: Focus on the Agent Execution System. Presented to the *Java Mobile Agents Workshop of the OOPSLA'97* (1997).
8. Rodrigues da Silva, A.: AgentSpace: Support, Development and Management of Agent-based Dynamic and Distributed Applications. PhD Thesis (in Portuguese), Technical University of Lisbon (1998).
9. Rodrigues da Silva, A., Delgado, J.: The Agent Pattern: A Design Pattern for Dynamic and Distributed Applications. Proceedings of the EuroPLoP'98, Third European Conference on Pattern Languages of Programming and Computing, Irsee, Germany (1998).
10. Rodrigues da Silva, A., Delgado, J.: AgentSpace versus Aglets: Agent Infrastructures for the Future Internet Applications (in Portuguese). Proceedings of the SBC'98, Congresso da Sociedade Brasileira de Computação, Belo Horizonte, Brasil (1998).
11. ObjectSpace: Voyager Core Package Technical Overview (1997)
12. ObjectSpace: Voyager and Agent Platforms Comparison (1997)
13. 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)
14. 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)
15. Riecken, D., editor: Special Issue: Intelligent Agents. Communications of the ACM (1994), 37(7)
16. M. Wooldridge, N. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2), Cambridge University Press (1995)
17. J. Baumann, C. Tschudin, J. Vitek, editors. Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems (Linz, Austria), Dpunkt (1996)
18. Milojicic, D. et al.: Concurrency, a Case Study in Remote Tasking and Distributed IPC. Proceedings of the 29th Annual Hawaii International Conference on System Sciences (1996)
19. Watson, M.: Intelligent Java Applications for the Internet and Intranets. Morgan Kaufmann Publishers (1997)
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley Longman (1995)
21. Rational Software Corp: UML – Unified Modeling Language, version 1.0 (1997) `http://www.rational.com/uml`