Improving Current Agent Support Systems: Focus on the Agent Execution System

Alberto Silva
Alberto.Silva@inesc.pt
INESC / Tecnical University of Lisbon

Miguel Mira da Silva
mms@dmat.uevora.pt
University of Évora

José Delgado
Jose.Delgado@inesc.pt
INESC / Tecnical University of Lisbon

Abstract

We propose in this paper a generic architecture to support, develop, manage, and interact with agent-based applications. In this context, we detail two main components of the architecture, namely: the AES (Agent Execution System) and the AEE (Agent Execution Environment). For both of them we describe their main concepts, data structures and interfaces.

We also propose a set of characteristics that we believe will also be present in the next-generation AES, such as: independence between AES and AEE, i.e., support to multi-language agents; execution places hierarchically organized; interaction mechanisms amongst AESs and agents based on the event driven model with potentially abortable events; and management and administration AESs capabilities.

Introduction

Nowadays, there are many ASS (mobile Agent Support Systems) proposals – such as Aglets [IBM97], Agent-Tcl [Gray95], Odysseya [GM97], Tacoma [JRS95] – that present a common set of functionalities and purposes. Nevertheless, they also present some important technical and even conceptual differences amongst themselves.

For instance, Telescript [Whi96] was the usual reference for ASS, with a very high technological level. However it is/was a proprietary system, with a difficult programming language, and not well adapted to a dynamic and open environment such as the Internet.

On the other hand, a system like the HTTP-based agent infrastructure [LDD95] is language and system independent, but shows severe limitations of the overall performance and difficulties in developing complex applications.

The Aglets Workbench, that represents Java-based ASSs, lacks an elaborate object model (e.g., without the notion of execution places hierarchically organized; without management operation on agent families and clusters) and technical capabilities (e.g., just two access control levels; without the notion of an agent classes manager).

We expect that in the next few years these ASSs would be progressively 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, we need better ASSs that incorporate the best features of existing ASS. For example, they should include the independence of HTTP-based ASS, the technology for migrating agents as found in Telescript, and the integration with Java as supported by the Aglets workbench.

In this paper we give a very small contribution on that direction by proposing a complete, integrated conceptual framework for a next-generation ASS. Instead of being a final product, the paper intends only to contribute to the definition of the future ASS's main characteristics. The focus of this paper is mainly on the ASS aspects. For a general overview of software agents and ABA the reader may consult a very wide related literature [GK95,BTV96,SMdSD97].

The paper is organized as follows. The next section presents in overview of the global architecture proposed for a next-generation ASS. We will then focus on the most important components: the agent execution environment in section 3, and the agent execution system in section 4. In section 5 we summarize the contributions with a small discussion in the Internet and Intranet contexts.

Global Architecture Overview

Our ASS proposal is divided into three major components: AES, for Agent Execution System; ACS, for Agent Class System; and AEE, for Agent Execution Environment (see figure 1).

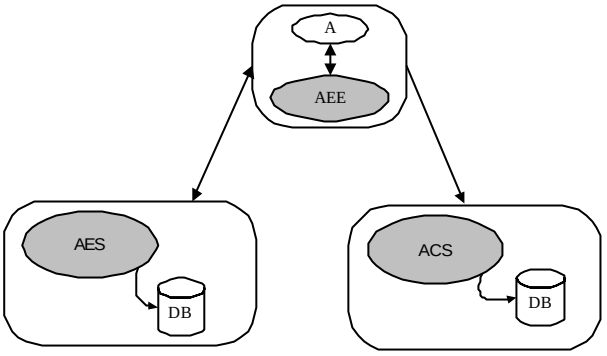

Figure 1: Main relationships between the major components in a ASS.

Several components may co-exist in the same node, several instances of the same component, or a node without a specific component. The only restriction is that an AES and an AEE instance co-exist in the same node in order to provide agent execution support (this in turn means that ACS is an optional component of the ASS).

Figure 2 shows the global view of the proposed architecture in more detail. The figure puts special emphasis on the multiple relationships between different components.

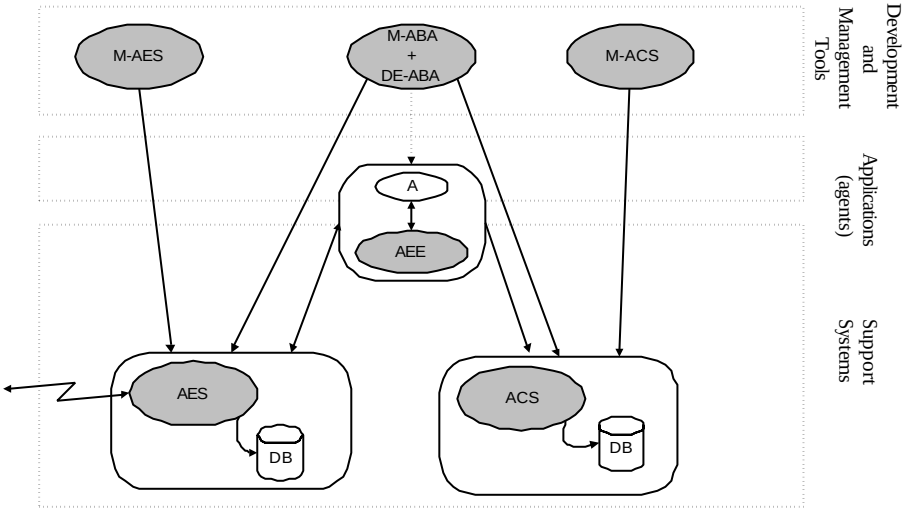

Figure 2: Global architecture overview.

In the lower level of the architecture there are the systems that support the storage and access to agent classes and their adequate execution.

- **AES (Agent Execution System)** – provides support to agent navigation and communication operations; provides persistence support; security; etc. The AES keeps its own database (or object store) where it tracks pertinent information about all agent and execution places (EP) supported, as well as data structures to support inter-agent communication. Additionally, the AES provides a set of communication protocols with similar and different AESs types.
- **ACS (Agent Class System)** – keeps (in its own internal database) a dynamic set of agent classes and provides a consistent and secure access interface, in order that other components (such as: M-ACS, M-ABA, DE-ABA, and even agents) may reference and instantiate, at run-time, a given set of classes.
- **AEE (Agent Execution Environment)** – provides an execution environment, corresponding typically to the interpreter or virtual machine attached to the programming language in which the agents were defined.

In the medium level there are agent-based applications that are executed by a virtual machine.

In the upper level, there are management and interaction tools as well as tools to develop ABAs.

- **M-AES (AES Manager)** – administrates one or more AESs. It should provide AES's configuration, management and maintenance capabilities.
- **M-ACS (ACS Manager)** – administrates one or more ACSs. It should provide ACS's configuration, management and maintenance capabilities.
- **M-ABA (ABA Manager)** – is an instance of an end-user specific application. It could be understood as a privileged interface between the user and its agents. It provides capabilities to manage an arbitrary set of agents, namely to create new ones, suspend, active, configure, etc.
- **DE-ABA (ABA Development Environment)** – supports the development, test and debug of agent classes and more generically of ABA. Ideally DE-ABAs should provide a powerful, complete and integrated environment, with several paradigms and techniques such as those found in the most modern tools (e.g., Visual Basic, Delphi, PowerBuilder, J++, etc.).

In this paper we focus our attention mainly on two of the different components identified: the AEE (section 3) and the AES (section 4).

AEE – Agent Execution Environment

The AEE is a virtual machine (VM) that allows the agent execution. With a given set of APIs (dynamic link libraries, packages, etc.) AEEs allow agents to access functionalities provided by AESs, ACS, or even different external resources. There is a strong relationship between the AES and the AEE, due to the fact the first restrict in fact the agent execution to the context of the AEE.

There is a possible variant of the conceptual model depicted on Figure 1 that is illustrated in Figure 3. This variant happens when the AES is itself supported by a VM (e.g., Java VM). In this case, and when the agents are executed by the same VM, the agents can be executed in the internal computational context of the AES as a thread of execution. In this case, the AES exported interface is comparatively easier to implement and more efficient, because it is based on direct method invocations within the AES/AEE process. We call "solution 1" to this concrete implementation of the proposed architecture.

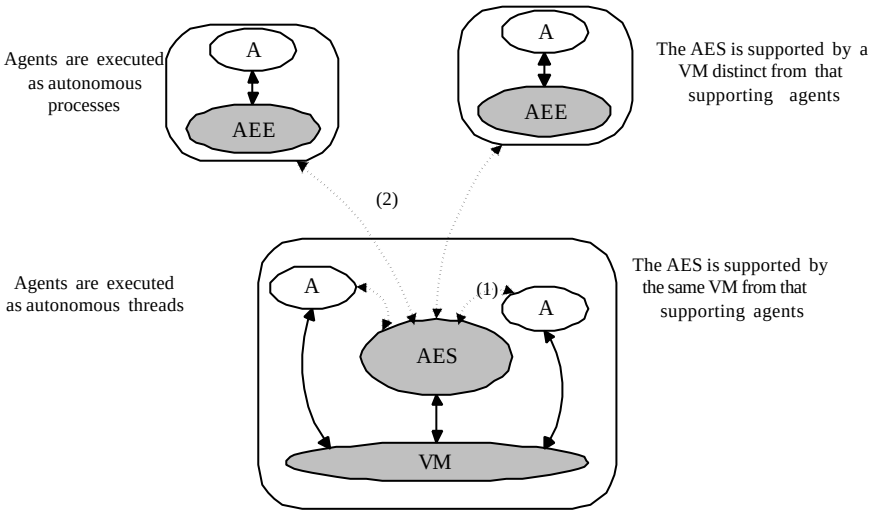

Figure 3: Possible relationships between the AES and the AEE.

On the other hand, the general solution requires a physical separation between the AES and AEE/agent processes. This is required, for example, to support several agents implemented in multiple programming languages. We call this variant of the basic architecture "solution 2". Although the interface AEE/AES is similar for both solutions, the implementation of solution 2 is harder to build and less efficient then its counterpart for solution 1.

It should be noted that any language can be used for building agents – not only Java – as far as the AES provides an equivalent API implementation. (However, we still prefer a simple, object-oriented, type-safe programming language for obvious reasons).

The AES is responsible for creating and dispatching the thread/process related to a given agent based on the information provided with the agent class information. Here we can distinguish two different situations:

- **Thread** – If the VM (AEE) supporting the agent class execution is compatible with the VM supporting the AES and the developed agent class is based on the provided AES's API, then this agent may be executed as an autonomous thread in the AES's internal computational context.
- **Process** – If the AES is not supported by a VM or if it is incompatible with that supporting the agent class, and at the same time there is an API implementation conformant with the AES's API specification, then this agent may be executed as a external process supported by the AES.

If none of the previous cases occur, then it is not possible to create an agent from the specified AES.

AES – Agent Execution System

The AES maintains a private data repository about EP (execution places), agents, end-users, groups, and their respective permissions.

The AES is typically a process executing multiple threads. Its main goal is to provide facilities that transport, store, manage, control and communicate with and between agents. As we mentioned in the previous section, the AES should provide the same capabilities whether the agent is being executed internally or externally to the AES context.

AES Class Diagram

Figure 4 shows in UML notation [Rat97] the main classes in the AES.

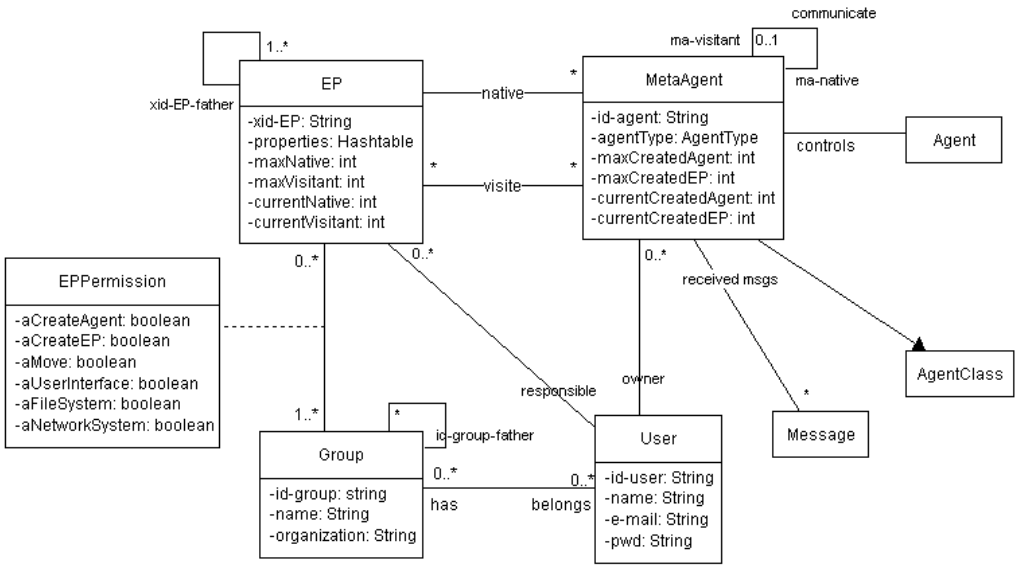

Figure 4: AES class diagram.

The EP (execution place) class 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 EP has a unique global identification (based on its AES's URL) and knows the identification of its user/manager. It also maintains a keyword/property list that allows an informal characterization. Optionally, the EPs can be hierarchically organized. The EP can also contain the maximum and current number of agents allowed in order to support elaborate resource management.

Each AES has at least one EP – called "default EP" – that is used every time an EP is not known or not even specified.

During its lifetime, a mobile agent can be executed in several EPs (either local or remote). An agent is visiting an EP when it is not executing in the EP where it was created originally. In order to keep track of its agents,

the EP keeps a list with its visitant agents and another with its native agents. The EP also knows in which EP its native agents are executing at a given point of time.

The MetaAgent class keeps agent-related information, namely: its own identification, its native EP identification; its end-user identification; the maximum and current number of created agents and EPs; and the identification of its agent class. Based on the agent class information, it is then possible to get all the messages (calling *getMsgs*) and public methods (*getInterfaces*) that the agent knows how to handle.

Additionally the MetaAgent class keeps the following information: the current EP where the agent is being executed; a list of received messages not yet handled by the agent; a flag telling if the agent is being executed internally (as a thread) or externally (as a process) regarding its current AES; and the AES internal identification.

When it is executing in their original AES, the agent has just one MetaAgent instance. However, when it is visiting another EP it has two MetaAgent instances – one in its native EP and the other in the currently visited EP. The instance in the visited EP provides a consistent and efficient access and management mechanism for the agent. The instance in the native EP helps to solve the "open channels" issue and provides local access and management capabilities. There should also exist a replication mechanism to keep the consistence between the original (kept in the native EP) and the temporary instance (kept in the visited EP) – for instance, based on the centralized proxy model.

Agents usually interact indirectly by exchanging messages via their MetaAgent instances (either local or remote).

The AES maintains lists of users and groups to implement the permission and control access mechanism. A user may belong to one or more groups. Groups may be hierarchically organized to simplify permissions. This means that all users of some specialized group have implicitly all the permissions they inherit from the more general groups, although the contrary is obviously not true.

AES Interfaces

Figure 5 shows the main interfaces provided by the AES, namely: interface with the AEE; low-level communication interfaces with other AES; the interface to be used by management and end-user tools; and interface with a database management system.

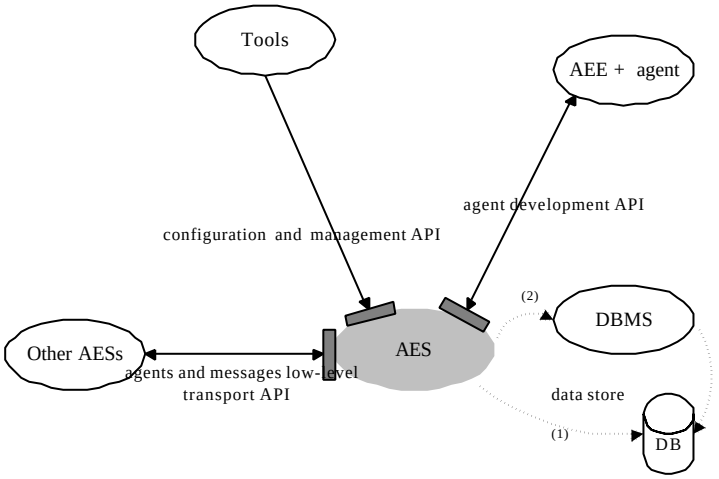

Figure 5: AES exported interfaces.

In the interface with the database management system, the AES works as the client of this interface and not as the server as in the other cases. (This is the reason why we depicted the arrows with a dotted line).

Interface with another AES

This interface permits to transfer agents and messages between two AESs and is based on a low-level communication protocol. This requirement makes this interface independent of higher-level but much less stable communication protocols such as RPC, RMI, IIOP, or DCOM. HTTP cannot be used, in general, because it assumes agents will live on the Web, when agents can have a life on the much more general Internet.

As an example, a communication interface based on the Java/RMI mechanism is described in example 1.

```
public interface TransportAES extends Remote {

  void acceptAgent (String idAgent,
     AccessTicket accessTicket, String idEPSource, String idEPTarget)
     throws RemoteException, InvalidPermission;

  Agent getAgent(String idAgent, AccessTicket accessTicket, String idEPSource)
     throws RemoteException, InvalidPermission;

  boolean sendMessage(Message msg, String idReceptorAgent)
     throws RemoteException, InvalidPermission;
  ...
}
```

Example 1: The interface with another AES.

The AES provides the following public methods:

- *acceptAgent* – invoked by the remote AES when it is about to send, to the current AES, some agent identified by *idAgent* and with accesses specified in the *accessTicket* parameter. The *idEPSource* and *idEPTarget* parameters specify the source and target EP respectively.
- *getAgent* – is invoked by the remote AES to extract, from the current AES, the agent specified by *idAgent* and executing in *idEPSource*.
- *SendMsg* – is invoked by the remote AES to send a message to an agent currently running or created in the current AES.

Figure 6 depicts the agent transfer mechanism between two ASEs based on the interface described above.

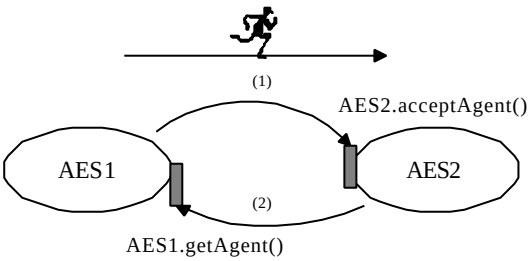

Figure 6: The agent transfer protocol based on Java/RMI mechanism.

Supposing that AES_1 needs to send an agent to another AES_2 then it first invokes the *acceptAgent* method in AES_2. The method call specifies the identification of the agent to transfer, the target EP, and the permition level. As a response to the previous invocation, AES_2 invokes *getAgent* in AES_1. As a result of this last invocation, AES_2 should obtain (in case there are no exceptions) a serialized version of the agent, and should be responsible by rebuilding it in the target AES. (This is a no trivial task, that involves several implementation decisions and sub-tasks, like for instance: loading the code closure of the involved agent; and creating and filling a MetaAgent instance with all relevant information.)

Interface with the AEE

The AEE interface is the main programming interface, to be used by agents and agent aplications alike. Agent applications use this interface to create, duplicate, and transfer agents. It can also be used to communicate between agent applications themselves.

EP (Execution Place)

The EP is responsible for accepting requests from agents and MetaAgents, its internal representatives. The EP is typically implemented as a thread of execution running within the AES. Agents can create other agents, but for the sake of simplicity, agents themselves cannot create EPs. (A new EP can only be created by using the management interface, see below.)

EPs are created with default permissions, and as such we propose to give them very limited default permissions. In alternative, the EP can inherit the permissions from the EP which created it.

MetaAgent

The MetaAgent class represents the main interface to manage agents. There is a MetaAgent instance for every agent in a given EP and it is responsible for hiding the details about the agent from other entities talking with that agent (for security or other reasons). For example, if the agent is being executed as an external process, then the MetaAgent is responsible for sending a message to that agent by calling a remote procedure on that agent. The same happens when the agent is being executed on another AES.

Agent

An agent can be executed in the same computational context (process) as the AES or externally as an independent process. The agent communicates with its AES via its internal representative, the MetaAgent. On the other hand, the agent itself has a list of callback methods the AES can use to send messages or requests to the agent. There is a defined callback for each event that can occur. The programmer can then change each callback depending on the particular application being built.

By using a callback before treating an event, agents can prevent an event from hapenning at all, generate new events, or replace an event by another event. For example, if the AES receives a "remove agent" event it should first call the agent to remove all its child agents before removing the agent itself. The agent can then check if it really should be removed in the *beforeRemove* callback, and if not, the agent can abort its own removal (provided the agent has permission to do that).

There are many other events and respective callbacks. The table below shows some of them, which can be extended by the application programmer as appropriate for a particular application. For each event there is a corresponding method in the AES invoked by the agent and another in the agent invoked by the AES.

Event	**AES supported method**	**Agent supported method**
Life cycle:		
- Create	*createAgent()*	*beforeCreate(); run()*
- Duplicate	*duplicateAgent ()*	*beforeDuplicate();afterDuplicate()*
- Suspend/Activate	*suspendAgent ()*	*beforeSuspend();beforeActivate()*
- Remove	*removeAgent ()*	*beforeRemove()*
Navigation:		
- by itself	*goTo()*	*beforeGo();afterArrive()*
- by a third entity	*come()*	*beforeComeBack(); afterArrive ()*
Communication:		
- synchronous and asynchronous	*sendMsg ()*	*handleMsg()*

Table 1: Events and methods related with the agent.

Agent Life cycle

The agent is created by calling the *createAgent* method in the EP executing on a given AES. The arguments for creating an agent include the AC (agent class) specifying the type of the agent and, as an option, a unique identifier for the new agent. The EP then creates a new MetaAgent responsible for this agent and all communication with it. The MetaAgent then calls *beforeCreate* and *run* to initialize the agent and start its execution. The *run* method is the main entry point to the agent and will be called every time an agent needs to be (re-) started.

Instead of creating an agent from scratch, an agent can also be created by duplicating (cloning) an existing agent of the same class (type) using *duplicateAgent*. In this case, the agent itself calls *duplicateAgent* that

creates a new agent with the same CA (agent class) but a different identifier, which can be passed as an argument or generated automatically by the AES. There are two aditional callbacks now: *beforeDuplicate* which is called in the original agent creating the replica; and *afterDuplicate* which is called in the new agent after the duplication.

The method *suspendAgent* stops the agent execution and stores its current execution context in the database. After a period of time, the agent is re-created and re-started. This period is passed as an argument in the *suspendAgent* method. There are two callbacks: *beforeSuspend* and *beforeActivate*.

Similarly, invoking the method *removeAgent* calls *beforeRemove* in the agent before it is actualy removed. (There is no *afterRemove* callback for obvious reasons.)

Agent Navigation

The agent can have the initiative to migrate to another PE by calling *goTo* on itself. The two callbacks *beforeGo* and *afterArrive* are called before and after migration, respectively. The agent can also be asked (or told) to come back to a EP by calling *come*, which has two associated callbacks: *beforeComeBack* and *afterArrive.*

It is an AES's responsibility to optimize the agent/message transfer mechanisms, such as the case of the agent navigation between EP supported by the same AES, i.e., between local EPs.

Agent Communication

Agents can also communicate with other agents by calling *sendMsg* which can be used either as asynchronous or as synchronous communication protocol. The receiving agent provides a callback *handleMsg* to receive messages sent to this particular agent.

The communication model is based on that proposed by the Aglets Workbench. Basically, aglets communicate via asynchronous messages that return "future replies". (These are based on "futures" and "promises" developed by Barbara Liskov as part of the Argus research project [Lisk88].)

In addition, we propose that all kinds of messages supported by a particular agent class – from which agents are created – should be registered in one or more ACSs.

Agents can also communicate directly by calling public methods on other agents. However, the following requirements should be met by the two agents: they are running within the same EP; they are running as a thread within the AES; and their public interfaces were exported by their respective agent classes.

Interface with Management Tools

The AES is configured, managed, and generally maintained by management tools that are just specialized agent-based applications. Typical operations include those involving:

- EP (native agents, visiting agents, and so on);
- users and user groups;
- permissions (relationships between EP and user groups).

These tools are also used to monitor and control agents and EP themselves, such as:

- answering questions about where are the agents (by EP, user, group) including those in remote EPs;
- suspending and activating agents (only local agents) by agent id, agent class, user, group, or EP;
- removing agents (including remote agents) by agent id, agent class, user, group, or EP;
- stopping and activating EPs.

In addition, there will be management tools for managing the AES itself, for example:

- starting (boot) and stopping (shutdown) the AES;
- managing and querying the event log;
- setting up communication ports;
- backing up and restoring the AES database.

These operations should be extended to remote AES so that a number of AES can all be managed centrally by only one system administrator.

Using these tools, an "agent system administrator" (that is, a real person) can easily have an overall picture of what is going on with agents under his or her responsibility and take actions as required. For example, it will be needed to start local and remote AES, create users and user groups, set-up and change permissions, create and stop agents, and so on.

Finally, the AES administrator will also need to manage the database, the local file system and potentially also the communication system. These will probably reside outside the AES, so there is a strong interaction between the operating system and the AES which needs to be better understood.

The management tools themselves can be implemented by agents or use any other technology. However, implementing them with agents will help users trust the AES and provide useful feedback for the AES provider as well.

Interface with DBMS

This interface to the DBMS enables the AES to save its internal data structures – agents, messages, and so on – persistently, reliably and safely.

This interface can be implemented in one of two basic approaches: either the AES keeps its own store with its own interface (represented by "1" in figure 5); or uses an existing DBMS via some standard API such as ODBC or JDBC (represented by "2" in figure 5).

Summary and Discussion

In this paper we proposed a general architecture for an agent support system with several interesting features. These include:

- the execution place as an execution and meeting point for agents, but also as a control mechanism for checking authorizations, access control, and resource management;
- users and user groups with permissions that are associated with execution places;
- support for one or multiple agent programming languages that permit application programmers to achieve the right balance between flexibility and efficiency;
- internally and externally executed agents, again permitting the right level of security and efficiency depending on the application characteristics;
- communication protocol between agents and their support system based on pre-defined methods and callbacks in both directions;
- a flexible but still simple event model that gives agents some control about themselves;
- a core set of classes and interfaces for accessing and managing agents and agent system.

Although quite straightforward, the architecture proposed in this paper is novel in several aspects. On one hand, it is a general framework not biased by any particular programming language as many other architecture proposals. On the other hand, it includes not only components for the agent system itself but also many other components that will be needed to manage and utilize the agent system.

In the future, we believe current agent systems will evolve to support many of the features proposed in this paper. However, before achieving that, a number of pragmatic decisions will have to be taken:

- Java and its libraries (RMI, JDBC, security, and so on) will be even more used for supporting not only agents but also agent support systems;
- many agent systems will support more than one agent programming language to take advantage of existing source code;
- the URL will continue to be used to identify agents and agent systems, even though it depends on the physical machine address (although agents can already be identified by any other mechanism);
- persistence will be based on external relational databases or the file system (with object serialization) even though agents require much more elaborate persistent support [MdSA96,MdS96,MdSS97].
- there will be many different communication mechanisms between agents, including synchronous and asynchronous, local and remote, direct and indirect, point-to-point and multicast, and so on;
- there will be many external interfaces to access resources not supported by the AES;
- there will be many other external functionalities (for example, access control) that programmers will have to write because existing ones cannot be integrated with an agent system.

The new JDK 1.1 for Java – with its libraries for communication, persistence and database access – provides an extensive support to build AES. (RMI is almost an agent system by itself!) However, each AES will use Java in its own way, and as a result a number of different, incompatible AES will appear. The new Mobile Agent Facility [Cry+97] from OMG can help to make different AES communicate, but it does nothing to normalise the AES itself.

If each AES offers its own programming model, then users will not be confident on agent technology and will have many troubles using a particular AES. It is an objective of this paper to help providers of agent system to build simple, integrated AES that can be easily understood and used by normal application programmers. In order to achieve that, we proposed a single, general, complete architecture for all AES.

References

[BTV96] J. Baumann, C. Tschudin, J. Vitek, editors. *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems (Linz, Austria)*. Dpunkt, 1996.

[Cry+97] Crystaliz Inc. et al. Mobile Agent Facility Specification (Joint Submission) - Draft 5. April, 1997.

[GK95] M. Genesereth, S. Ketchpel. *Software Agents*. Standford University, 1995. `http://logic.standford.edu/sharing/papers/agents.ps`

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

[Gray95] R. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM'95 Workshop On Intelligent Information Agents*. 1995.

[IBM97] IBM Tokyo Research Laboratory. The Aglets Workbench: Programming Mobile Agents in Java, 1997. `http://www.ibm.co.jp/trl/aglets`

[JRS95] D. Hohansen, R. Renesse, F. Schneider. *An Introduction to the TACOMA Distributed System.* Computer Science Technical Report 95-23. University of Tromso, Norway, 1995.

[LDD95] A. Lingnau, O. Drobnik, P. Domel. An HTTP-Based Infrastructure for Mobile Agents. In *Proceedings of the Fourth Int'l WWW Conference*, 1995.

[Lisk88] B. Liskov. Distributed Programming in Argus. In Communications of ACM, 31(3):300-312. March 1988.

[MdSA96] M. Mira da Silva, M. Atkinson. Combining mobile agents with persistent systems: Oportunities and challanges. In Baumann et al. [BTV96].

[MdS96] M. Mira da Silva. *Models of Higher-order, Type-safe, Distributed Computation over Autonomous Persistent Object Stores*. PhD Thesis, University of Glasgow, 1996.

[MdSS97] M. Mira da Silva, A. Silva. Insisting on Persistent Mobile Agent Systems with an Example Application Area. In *Proceedings of the Mobile Agent'97 Workshop*. 1997.

[Rat97] Rational Software Corp. UML – Unified Modeling Language, version 1.0. 1997. `http://www.rational.com/uml`

[SMdSD97] A. Silva, M. Mira da Silva, J. Delgado. AgentSpace: 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.

[Whi96] J. White. General Magic, Inc. Mobile Agents White Paper. `http://www.genmagic.com/agents/Whitepaper/whitepaper.html`