Submitted for publication, please do not reference or distribute!

Mobile Objects and Agents (MOA)

Dejan S. Milojičić, William LaForge and Deepika Chauhan

The Open Group Research Institute

ABSTRACT

This paper describes the design and implementation of the Mobile Objects and Agents (MOA) project at the Open Group Research Institute. MOA was designed to support migration, communication and control of agents. It was implemented on top of the Java Virtual Machine, without any modifications to it. The initial project goals were to support communication across agent migration, as a means for collaborative work; and to provide extensive resource control, as a basic support for countering denial of service attacks. In the course of the project we added two further goals: compliance with the Java Beans component model which provides for additional configurability and customization of agent system and agent applications; and interoperability which allows cooperation with other agent systems has been adopted most recently.

This paper analyzes the architecture of MOA, in particular the support for mobility, naming and locating, communication, and resource management. Object and component models of MOA are discussed and some implementation details described. We summarize the lessons learned while developing and implementing MOA and compare it to related work.[1]

1 Introduction

Mobility has always attracted researchers in computer science. This interest spans from general observations, such as *"if it weren't for mobility, we would still be trees"* [10], and the analogies with the real world *"migrating birds and nomadic tribes moving due to the lack of resources"*, to purely technical reasons, such as improving locality of reference and difference between local and remote semantics.

One of the first incarnations of software mobile entities were called worms [32], which could spread themselves across nodes and arbitrarily clone. Unrestricted implementations of worms and viruses have received negative connotations, due to the attacks performed on networks [13].

The next generation of mobile entities, known as process migration, were implemented at the operating system (OS) level. There were many implementations of process and object migration [3, 12, 20, 33], but none has achieved wide acceptance. Due to inherent complexity it was hard to introduce process migration without impacting the stability and robustness of the underlying OS.

Mobile objects and agents have attracted significant attention recently. Unlike mobile code (such as applets), they consist of mobile code, data and non-transient system state that can travel between the nodes in a distributed system (intranet or Internet). Compared to mobile objects, mobile agents also represent someone; they can perform autonomous actions on behalf of a user or another agent. In the rest of the text we shall use the term mobile agents. A number of academic systems (such as Agent Tcl [21], Mole [26], Ara [29] and Tacoma [19]) and industrial systems (such as Telescript [36], Aglets [2], Concordia [9] and Voyager [35]) exist. The products using mobile agents start to appear, such as Guideware [17]. The government is interested in funding work on agents [11]. A patent has been approved on mobile agents [37], and a standard is in progress, OMG MAF [28]. A couple of books have been published on agents [5, 8] and a few of them is in progress [22, 25].

This paper describes the Mobile Objects and Agents (MOA) project at the Open Group Research Institute. The obvious question is why yet another mobile agent system? There were a few reasons. None of the existing systems at the time of starting the project were mature enough to be used as a starting point for our work. We found it easier to develop another system that would suit our needs from the beginning. Additionally, some areas of our interest, such as communication and resource control are deeply involved in the design decisions of any system, making it very hard to add them as an afterthought. Finally, we were interested in interoperability between the systems, and therefore supporting another implementation was a good idea anyway.

At the beginning of the project we were interested in the first two of four features listed below. During the course of development we added the additional two:

Collaboration. Frequently, agents need to collaborate during their execution either with other agents or their user. For

1. This work was supported in part by the Advanced Research Projects Agency and the Rome Laboratory of the Air Force Materiel Command.

agent collaboration, it is required to support naming, locating and communication among agents.

Denial of service attacks. Agents, as well as hosts, are vulnerable to mutual attacks, either over network or locally. In order to prevent denial of service attacks, it is required to maintain resource control of agents and agent systems, and to impose security and resource policies.

Configurability and customization. It is increasingly difficult to configure and customize software. In case of mobile agents, this applies both to agent applications, as well as to agent systems. Being compliant with a component model, such as Java Beans, allows for a standardized way to access and change component properties.

Interoperability. Agents, as well as agent systems need to interoperate. In case of agent systems, interoperability leads to a larger base that agents can visit. We were active in the OMG Mobile Agent Facility proposal which addresses mobile agents systems interoperability [28].

More details on how these goals have been achieved is described in Sections 4.3, 4.6, 4.2 and 4.2 respectively.

The rest of this paper is organized as follows. In Section 2 we provide a background on mobile agents and component based computing. Section 3 describes Java's suitability for mobile agents and for component based computing. Section 4 presents the MOA design and implementation. Section 5 discusses MOA current status. Section 6 describes some of MOA applications. In Section 7 we present lessons learned while designing and implementing MOA. MOA is compared to related work in Section 8. Finally conclusions and future work are presented in Section 9.

2 Background

In this section, we provide background on Mobile agents and component based computing.

2.1 Mobile Agents

Among multiple benefits of mobile agents we would like to underline the following. **Improving locality of reference** is achieved by moving the action towards the source of data, or other end point of communication. This way, substantial performance improvement can be achieved in case of slow communication lines or excessive amount of data transferred. **Survivability:** similar to nomadic tribes or migratory birds, agents can survive if moved closer to resources, or away from partially failed nodes. This is applicable in large systems where other approaches require complex solutions. **Analogy to the real world** helps some programmers to better understand programming paradigms expressed in terms of mobile agents. Examples are travelling salesman, shoppers and workflow management systems. **Customization** of software can be achieved using mobile agents, for example, by adjusting the search according to a user-specific criteria, or by performing an action specific to a remote site. Even though similar benefits are typical for mobile code, agents bring further advantage by being able to clone and visit multiple node autonomously. **Autonomicity** represents agent's independence from its owner. A user can start an agent and disconnect. While user is disconnected, the agent acts on his behalf. When the user reconnects, the agent returns or otherwise provides results.

Agents have various areas of deployment, such as the following. (More details on agent benefits and deployment can be found in [7].) **Slow and unreliable links**, such as radio communication, where locality of reference improves performance, and avoids potential loss while transferring large amounts of data. **Software distribution** becomes increasingly hard. Mobile code has provided a revolutionary breakthrough, by allowing downloading code for heterogeneous environment. Mobile agents makes this effort even easier, by associating actions and state with each distributed version and copy of a particular software. **Network management**: agents migrate both code and data, making them useful for automating control and configuration in large scale environments, such as networks [15]**. Electronic commerce** deploys mobile agents by modeling travelling salesmen or shoppers visiting stores in an electronic mall. **Data mining** is convenient application for mobile agents due to locality of reference: agents optimize search by wandering from a site to site with large volumes of information.

Nevertheless, mobile agents still haven't achieved wide acceptance. Technical solutions to many problems have been found; some others are currently being investigated. The success of this area mainly relies on the following unsolved problems. **Lack of applications**: similar to other forms of mobility, mobile agents have achieved a reputation of the solution searching for the problem. Many systems have been developed but few applications exist. **Security:** the problems caused by mobile code are frequently reported. Mobile agents push the security problems even further. **Lack of infrastructure** well suited for mobile entities**,** such as name servers, messaging systems, and commands to be delivered to moving targets, is still not widely deployed. **Survivability** offers both benefits and challenges for mobile agents. Mobile agents are inherently survivable, but this support does not come for free; mobile agents need to be designed and implemented for survivability. In particular, they should minimize residual dependencies on previously visited nodes, or a centralized server.

2.2 Component Based Computing

Component based programming, including OpenDoc, VBX, and ActiveX, has been quite successful in speeding the development of GUI applications. With component programming freed from its close tie to the GUI, the same building block approach can be applied to non-GUI development. Java Beans (components written in Java), show particular promise for non-GUI component programming.

The runtime behavior of a Java Bean is defined by an ordinary Java class. The difference between a bean and other objects is the metadata used for configuration. This metadata is provided by an associated BeanInfo class, or it is derived from the runtime class.

Component-based programming enhances object oriented benefits, such as flexibility, and code reuse with two new characteristics: independence and configuration.

Independence. The source code defining a component does not directly reference any other component; instead, relationships between components are created at runtime. The relationships may be established by the container holding the components, or even by the component itself. This has several benefits:

- a "building block" approach: programs are constructed from existing components by defining relationships
- each component can be individually tested
- components are more easily reused, there is a minimum of interdependency between the components
- a program can be restructured to meet new requirements without impacting the logic of individual components
- updating a program with the latest version of 3rd party components is simplified

Configuration. A component is constructed by a general configuration tool. The component participates in its own configuration. Application programs are assembled from the pre-configured components. The implementation specifics of a component are kept from all other elements of the program. Separating the configuration of components from an application program has several benefits:

- separation of configuration from the application programs facilitates the use of alternative implementations
- component upgrades are backward compatible
- application-specific logic governing the configuration data is separated from the rest of the program

Separating component configuration from an application program does impact the development cycle of a program, as changes made to a component's source code will often invalidate the configuration of that component. The edit, compile, and test of the development cycle now becomes edit, compile, configure, and test.

3 Java

We have chosen Java as a target because it seemed to carry the mainstream development, and also because of the features that make it suitable for mobile agents. During development we discovered that it is convenient for components.

3.1 Java and Mobile Agents

Java offers advantages for mobile agents, as well as some disadvantages. Advantages consist of the support for mobile code, heterogeneity, language safety, object serialization [30], reflection, dynamic class loading, and multithreading.

Disadvantages consist of inadequate support for resource management (no support for managing memory and disk limits), no support for preserving the thread execution context, limited support for versioning (will be addressed in the forthcoming releases), no ownership of objects and fine grained protection at the object granularity.

3.2 Java and Components

Java is a particularly appropriate language for components. Components written in Java are lightweight; little code is required for conformance to the component model. Java supports a number of key features of component programming:

- A component may have several interfaces. Java provides for the implementation of multiple interfaces, unrestricted casting, and an `instanceof` operator to determine if a component supports an interface.
- Several components may be aggregated into a single component. The JDK 1.1 methods `Beans.isInstanceOf` and `Beans.getInstanceOf` can be used in place of the instanceof operator and casting, allowing for the future use of aggregation in JDK 1.2.
- The life of a component may span more than one program. JDK 1.1's provision for serializing components allows converting an object into a form which can be written to a disk file or passed across a network.
- A component is configured by modifying its properties. The properties are identified by examining the method signatures of the component. For example, the class is recognized as having the property `slices`, by virtue of the presence of the `getSlices` and `setSlices` methods.
- The component's properties are accessed using the JDK 1.1 class Introspector.

4 MOA Design and Implementation

The MOA architecture is presented in Figure 1. There are three types of nodes that run MOA system: front end node allows users to control and monitor agents; home node is used as a repository for agent's data; and remote node is where agents typically run throughout their life time. The MOA system has a Telescript-like model (although sufficient difference avoids infringing their patent). There are agents that travel visiting places that are held by Agent Environments (AE). Places serve to accept agents, and to store

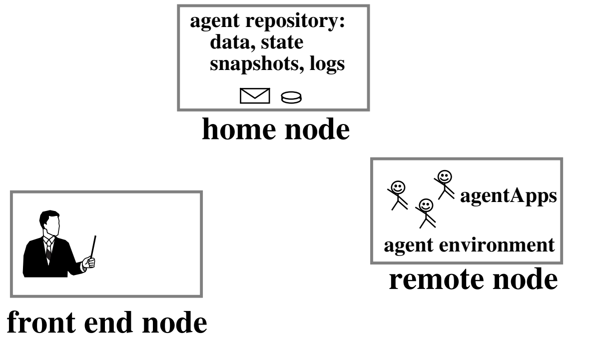

Figure 1: MOA Configurations: *there are three different MOA configurations. Front end allows user to start and control agents and other MOA components. Home node is the node where agent was originally started and where agent-related state is maintained. Remote node is one of the nodes where agent currently executes.*

information. Agent environments host various objects. A name server tracks the location of agents and potentially other objects, whereas a monitor serves for controlling and monitoring objects. These and other objects in the MOA are described in more detail in this section.

4.1 Object Model

The MOA objects for remote nodes are presented in Figure 2. They can be classified as agent- or system-related. Agent related objects are circled; they have migratory state. Agent and place belong to the user trust domain (see Section 4.7 on more details related to security), whereas other components belong to the MOA trust domain.

An agent and place are application extended classes. **Agent** is the first class object in MOA. There is a template class that is extended by agent application. Agents are named (see Section 4.4 for agent naming), and they can communicate. An agent can perform actions, such as *move* to a certain agent environment (or a place within it), request to *meet* with other agents at a certain place or agent environment, *openChannel* to another agent, or *sendMessages* to it. An agent always executes within a place (see below). There is a one-to-one mapping between an agent and a place within an agent system. However, an agent can leave places behind when it moves. Therefore there is one-to-many mapping between an agent and places on many agent systems.

Place is the second class object in MOA. The main difference from agent is that place is a stationary object. Therefore a place can not *move* and *meet*. However, places can communicate with other places and agents; they can be active, i.e. they can have threads running. Place also serves the container-proxy role. They are proxies because they can remain after an agent leaves and represent it there (be proxy). They serve the container role for security and resources of an agent.

Agent Control, Agent Properties, Family, LogService and Bucket are agent related classes that belong to the MOA trust domain. **Agent control** is an internal class that represents an

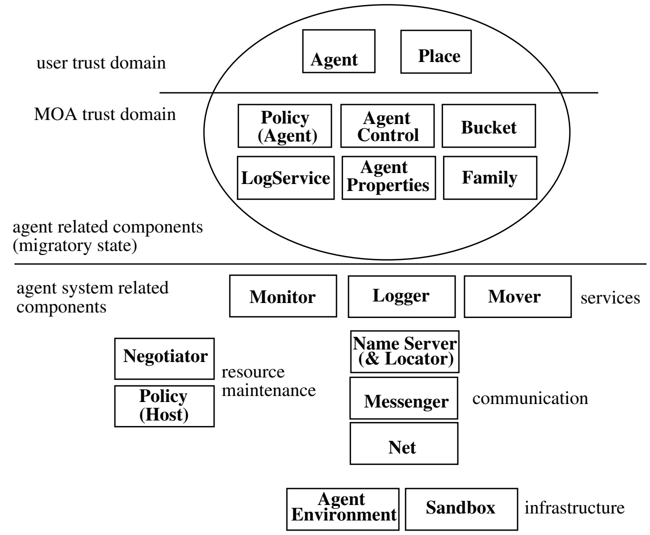

Figure 2: MOA Objects: *consist of objects in user trust domain, and MOA objects in MOA trust domain. Agent environment represents container for all MOA objects.*

interface between agent system and agent/place. It manages agent system resources (communication channels, agent properties, etc.). These objects can be accessed by an agent/place, but they cannot be changed. **AgentProperties** contains all the properties that characterize agents and are transferred across the nodes. Examples are owner, home Agent Environment, locatingStrategy, etc. **Family** is used for monitoring an agent's activities. Each agent has its own Family object which it carries across migrations. **LogService** is a component managing local logs. **Bucket** holds the contents of a JAR file (JAva aRchive). Each bucket implements a classloader for the dynamic loading of the jar file. A hashtable contained within a bucket enables the client of a bucket to efficiently index into the contents of a jar file. During migration, all components circled in Figure 2 are serialized, put into the bucket and sent to destination node.

Policy and Negotiator objects maintain and manage information about resources. **Policy** is a placeholder for properties describing the policy of an agent arriving at a node (agentPolicy), and a host receiving the agent (hostPolicy), such as agent's maximum lifetime, maximum number of channels and maximum threads. **Negotiator** performs negotiation between the agent and receiving agent environment prior to agents visit. Agent movement is subject to resource requirements and security arrangements between the two entities.

Sandbox and Agent Environment provide basic infrastructure. **Sandbox** class separates the agent application from the agent system state. It switches from the agent system thread to application thread when there is a different protection domain; it also serves to switch from synchronous to asynchronous communication when going across the network. Resource usage and limits are tracked on a per sandbox basis. **AgentEnvironment** (AE) is the container for agents and

- root
  - **Net** (Net)
  - **Messenger** (Messenger)
  - **Logger** (Logger)
  - **HostPolicy** (Policy)
  - **Mover** (Mover)
  - **Killer** (Killer)
  - **NameServer** (NameServer)
  - **Monitor** (Monitor)
  - **AcTakedownMgr** (AcTakedownMgr)
  - **AE** (Environment)
    - **sam:457#Third_1** (AC)
      - **AcChannelMgr** (AcChannelMgr)
      - **AcMsgUser** (AcMsgUser)
      - **Agent** (application-specific, derived from Agent)
      - **Place** (application-specific, derived from Place)
      - **AgentProperties** (AgentProperties)
      - **LocalPolicy** (Policy)
      - **Family** (Family)

Figure 3: Agent System Object Tree: *defines the names of MOA internal objects. In parenthesis we provide the class names from which the objects are derived. Internal object names are important when communication between various objects on different MOA systems is established, and for initialization.*

- class sandbox.environment.**EnvironmentChild** (implements java.io.Serializable)
  - class moa.ac.**AcTakedownMgr** (implements moa.mover.AgentStateListener, java.lang.Runnable)
  - class sandbox.environment.**Environment**
    - class moa.ac.AC
  - class sandbox.environment.**Killer** (implements java.lang.Runnable)
  - class moa.ac.**LogService**
  - class sandbox.message.**MessageUser**
    - class moa.nameserver.**LocatorUser**
      - class moa.ac.**AcMsgUser**
      - class moa.nameserver.**NameServer** (implements moa.nameserver.NameService, sandbox.message.DeadMessageReceiver)
        - class moa.nameserver.**HNameServer** (implements moa.nameserver.HNameService)
      - class moa.user.**User** (implements java.io.Serializable)
    - class moa.logger.**Logger**
    - class moa.monitor.**Monitor** (implements sandbox.environment.EnvironmentListener, moa.mover.AgentStateListener)
  - class moa.mover.**Mover**
  - class sandbox.net.**Net** (implements sandbox.CallBackListener)
  - class sandbox.net.**NetUser** (implements sandbox.CallBackListener)
    - class moa.ac.**AcChannelMgr**
    - class sandbox.message.**ChannelManager**
      - class sandbox.message.**Messenger**
  - class moa.api.**Service**

Figure 4: Inheritance tree of EnvironmentChild Class: *the EnvironmentChild class provides methods for accessing other components within the object tree (see Figure 3). For example, the Mover needs access to NameServer and Messenger and therefore has to be wired. Inheritance tree indicates objects accessible by inheritance, for example, Mover has access to Messenger by being a subclass of MessageUser, and therefore does not need to be wired.*

their related objects at an agent system. There is one agent environment per Java Virtual Machine (JVM), but there can be many per a node. Each agent has its home AE and alternate home AE in case the home is not accessible.

Net, Messenger and Name server comprise the MOA communication model. **Net** provides the basic communication support for establishing and maintaining communication channels between components on remote and local JVMs. Communication channels are established by specifying the component name, host and port number. The **Messenger** layer uses the services of Net to support one- and two-way messages between components. Components are addressed using destination agent system and the component name. **Name server** tracks agent locations. The name server clients can *(un)register* and *lookup* an agent location. Name server clients are user (monitor) and agents that *lookup* locations of other agents in order to communicate. Information about the agent location (or how to find it) is cached at agent systems that the agent visited or communicated with. Name server also plays the **Locator** role.

Mover, Monitor and Logger are services that MOA supports. **Mover** performs the moving activity between nodes. It negotiates migration, captures the agent state, and transfers it. **Monitor** provides a user interface to monitor and control agent applications. A user can be informed about the agent's movement, resource usage, and other statistical information. **Logger** logs events in an Agent System to persistent media.

MoaApplet, BatchDriver and User classes support the interface to the MOA system and its applications. **MoaApplet** is an applet-based interface enabling users to interactively monitor and debug agents, to launch them, to snapshot the agent's state, and to query the logged data. **BatchDriver** is a script based user interface to provide the services of MoaApplet; typically it is used for testing purposes. **User** class serves as an interface between the user applet and the agent. It launches the preconfigured agents, tracks the agents, and maintains information of interest to a user.

4.2 Component Model

All MOA components are configured using the MOAbatch tool (see description in Section 5). Configuration of the components of the agent system effectively defines the object tree of the system. The MOA system is loaded by first loading the root component. Each component then successively loads the component below it in the tree. Components are locally organized into a labeled tree (see Figure 3) used to dynamically establish relationship between the components, in contrast to static binding, typical of OO programming.

After the components have been loaded, the initialization process permits the components to locate other components of the agent system by name in order to establish dynamic binding. Non-leaf elements of the object tree subclass environmentChild (see Figure 4) which provides methods for locating a tree element given a relative or full pathname. For example, any component in Figure 4 can access the Mover object by calling the method:

```
(Mover)getEnvironment().getInstanceOf("/Mover",Mover.class)
```

Coupling of components to a certain extent negates the benefits of component programming, and as such it has been kept to a minimum. For example, the net component is made

Debug (List of components that are being debugged)
AE (agent application components)
Batchdriver/MOAApplet
Messenger (netBean, timeout)
Monitor (messengerBean, timeout)
Mover (messengerBean, timeout)
Net (retryDelays(Increment, max, initial) runServer, host, port, exitServerOnError)
Home Service (nameServerBean, messengerBean, nameServerNetBean)
Logger (fileSuffix, logPath, retentionPeriod, filePrefix, messengerBean, timeout)
Name Server (nameServerBean, messengerBean, nameServerNetBean,)
Users (userPath,nameServerBean,passwd,messengerBean,nameServerNetBean,HSretryDelay)
UE (listOfUserBeans)
Host Policy (maxLifeTime,timeRemaining,maxChannel,remainingPlaces,maxThreads
Killer (killerTimeout)
Root (componentNames: AE,debugger,net,messenger,logger,hostPolicy,mover,nameserver,
UE,homeServer,monitor,ACTakeDownMgr,killer)
ACChannel Manager (timeout, netBean)
AcMsgUser (nameServerBean,messengerBean,nameServerNetBean,timeout)
AC (AcChannelMgr,AcMsgUser)

Figure 5: MOA Components *are described with the list of properties. This figure lists configuration of the remote MOA system.*

known to other components, such as those that subclass NetUser: AC, Messenger and arbitrary applications using Net.

We have successfully used components for preconfiguring agent applications, as well as the agent system. For agent applications, we can easily preconfigure an agent's itinerary, policy, types of logging, debugging, etc. Agent system configuration specifies which components will be integrated into the tree. For example, homeService is not present on all agent systems, agent environment is not needed on the front end and on the home node. Debugging can be specified as a part of the agent system configuration at class granularity. Message timeouts can be configured on a component basis. For the Net component we specify its port number; for each agent, we specify the number of service threads is specified for each agent; for the Killer component we indicate the time when it will take the system down; for each user, we specify the password, login id (equal as the component id) and login time; for each system, we specify the host policy for negotiating with agents.

Components can also be organized using subcomponents. For example, AC is an aggregate which includes AcChannelMgr and AcMsgUser. Only AC is configured into the whole system. This way configuration is simplified using a hierarchical structure. The MOA component model of the agent system is presented in Figure 5.

4.3 Communication

Communication support in MOA is built on top of JVM support for pipes and sockets. It provides a higher level of abstraction, such as the channels and messaging between MOA objects (agents, places and servers). The communication channels support object streams. Messaging provides for passing objects of arbitrary type specific to the application.

The Net package supports opening of channels with automated retry. The destination system can optionally reject a request for the channel subject to resource limitations. This can happen at the agent system, as well as at the application level. When opening a channel to an agent, only the agent name needs to be specified. The agent system resolves the actual agent location with the help of Locator object in a distributed manner.

Locator
(AcMsgUser, NameServer, User)

Messenger
(LocatorUser, Monitor, Mover, Logger)

Net
(AcChnlMgr, ChannelMgr)

Figure 6: Communications Layers: *in MOA are stacked. Net layer supports stream based communication, messenger supports messaging and locator introduces transparent locating of migratory objects.*

The Message package is able to pass application specific objects by delaying deserialization until the name space of destination is identified. Messages can be synchronous (RPC-like), or asynchronous (one-way messages). Messaging is built on top of the Net package, using a common pool of channels dedicated to message passing. The destination of messages can either be a specific location (destination name, host and port), or a logical name (e.g. agent name) in the case where the destination is a migratable object. This is reflected in the implementation, where a layered approach is applied by building the Locator on top of the Message layer which builds on top of the Net (see Figure 6). The Locator handles transparent routing of messages when a location is not specified. It enables the agents to transparently communicate and collaborate with each other by just using the name of the agent (as the agents are migrating, their location at any given point of time is not known). It uses the locating strategies described in Section 4.4 to find new agent location.

In the case of two way messages, responses are routed back to the originating thread, which is suspended pending either a response or a time out. A response can arrive from the node other than the original destination, if the destination agent moved.

When moving from one node to another, the agent does not notice that its channels have been closed and then re-opened on the remote node. The underlying agent system achieves this transparently to the agent. When an agent transfer is initiated, first the channel migration process is performed. During this process, the agent informs its collaborating partners of its intent. From this point onwards, the data received on the channels is not passed to the application, but is stored in a Vector of unread Objects. Upon learning about the move, the agent channel manager on the other end of the channel reply back an acknowledgment and close their channel socket without informing the application. For the migrating agent, when the acknowledgment is received on a channel, the Reader thread for the channel is stopped, and the socket is closed. The agent transports itself to a new location, only when this migration process is completed for all open channels.

AEname *(ae)*:	***h:p***	***h*** - host name
AgentName *(a)*:	$ae_{home}\#f\_l.g$	***p*** - port number
PlaceName:	$a_{owner}\%ae_{residing}$	***f*** - family name
ServerName:	***h:p***	***l*** -launch number
		g - generation number

Figure 7: MOA Named Objects: *Agent Environment, Agent, Place and Server*

During channel migration, though the socket is closed, the information regarding the other communicating agents/places is still maintained. When the agent moves, it carries along the information about the channels, and uses it to reopen channels at the new location. Prior to reopening channels it first sends all the unread objects to the application. Reopening of channels can be done either eagerly or lazily, depending on the type of the application. If there are many channels that are not used very often, then they are re-opened lazily. If there are a few channels that will most likely be used promptly after migration, then they are reopened eagerly.

MOA uses remote method invocation within a number of its components. It would have been a reasonable choice to use Javasoft's RMI support for this, but we have not made such a decision. There is nothing to prevent us from using it in the future (and most likely we shall switch to using it), but for the initial implementation we did not want to adopt yet another technology (in addition to Java Beans) that would introduce learning curve. Furthermore the unclear situation with CORBA v. RMI was another contributing factor. Instead of using RMI, we simulate its functionality by sending objects across the network; based on the object type we invoke appropriate methods. The actual implementation is trivial. There are a small number of uses of remote method invocation and these are confined to limited scope of the MOA implementation, whereas MOA applications can use RMI. Overall, it was more a political rather than a technical decision not to use RMI for the initial implementation.

4.4 Naming and Locating

The following objects are named in a MOA system: agent environment, agent, place and servers. Name syntax is presented in Figure 7. An agent environment is named after the node name and port on which they perform communication. A server name has similar syntax.

Agents are named after the AE where they were created if they are first generation. If cloned, agents are named after the AE of their first ancestor extended with the generation number (see Figure 8), irrespective of the AE where they were actually cloned. In other words, each agent bears the sign of the original site responsible for initiating the agent family. This is used as an ultimate source of information on the current agent location: as a last resort, the home AE can be queried for current location information. Place name consists of the name of an agent it belongs to, extended with the AE name where place currently resides.

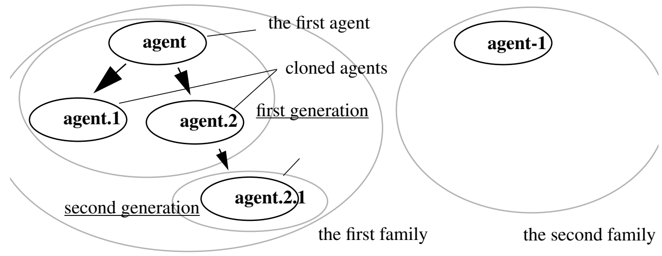

Figure 8: Naming of agents: *agent names are organized around agent families and generations. Cloned agents always carry the name of its ancestor as a part of their name.*

The agent location may be needed by its owner explicitly in order to track the agent location (request by the agent owner), or implicitly in order to be able to control it (*kill*, *suspend*, *resume*, etc.). It is also needed by agents in order to be able to communicate to (open channels) or synchronize (propose meeting) with other agents.

Locating agents is performed through name servers. Application requests to name servers (*lookup, register, unregister*) are issued through the agent environment which performs security and consistency verifications. When migrating the agent, the Mover object makes a local request to the Name Server. When opening channels or sending messages, NetUser and Messenger interact with the Name Server. Name servers on multiple nodes then cooperate to satisfy these requests.

The location object contains either the current location of an agent, or sufficient information to obtain it. In particular, it contains some or all of the following: the name of the residing agent system (if known), the type of the strategy to locate the agent (discussed below), the list of the nodes where the agent may reside (*itinerary*), the lifetime of this location object. Even though the lifetime of an agent is limited by its owner, because of the delays in transferring the agent over network, it is not possible to assure its accuracy.

The location object is cached at each node the agent visited, or where there was a channel opened with the agent. When searching for an agent, the location object is first looked up at a local name server. If not found, it is looked up at name servers higher in the hierarchy, if any exists (name servers may be organized in a tree-like hierarchy). If the agent is still not found, then the agent's home node is approached. The home agent environment is the ultimate source of information of agent location. The location of the home agent environment is implicitly known from the agent's name.

When locating agents different location schemes are used, similarly to those used in distributed operating systems, such as Charlotte [3], V kernel [33], Sprite [12] and Mach [24]. The MOA system supports: a) **updating** the home location of every agent move, b) **registering** at a predefined name server, c) **searching** based on predefined itinerary and d) **forwarding** based on the trails left during its migration. Locating

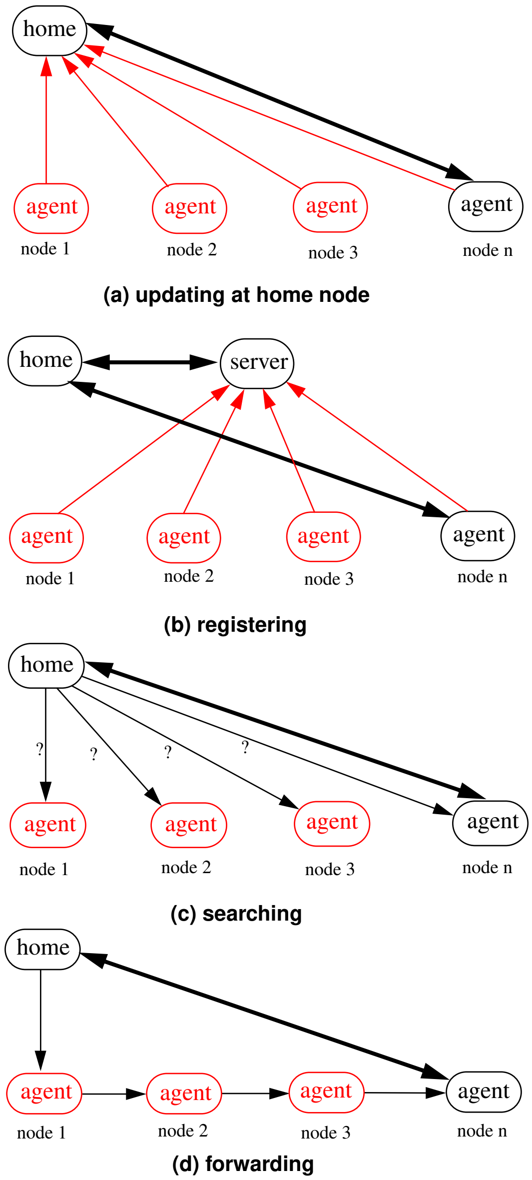

Figure 9: Locating schemes: *(a)* ***updating****: The agent always updates its location with the home node name server; (b)* ***registering****: agent always registers at a predefined name server; (c)* ***searching****: based on available itinerary, the sites are searched for the current agent location; (d)* ***forwarding*** *based on trails agent left behind.*

schemes are presented in Figure 9. Locating scheme is selected subject to:

- destinations (a local or a far away region; limited set of destination hosts or unknown),
- security aspects (if agent crosses security domain)
- type and frequency of migration (burst migration, or sporadic; few or many; frequency; cyclic).

For example, updating the home node is suitable for an agent that moves within a local region, it is not suitable for agents that visit distant nodes. Registering is suitable for an agent

agent properties (owner, familyName, home & alternateHome AE, lifeTime)
agent policy (maxLifeTime,timeRemaining,maxChannel,remainingPlaces,maxThreads)
family (typesOfLog, tracing, watches, limits)
agent (applicationSpecific)
place (applicationSpecific)

Figure 10: MOA agent applications Component Model: *in parenthesis we present configurable properties.*

that migrates within a far away region; in case of large number of nodes, registering nodes are organized in a hierarchical manner; it is not suitable for large number of migrations. Searching scheme is suitable for agents that visit a small number of known hosts; it is not suitable for destinations not known in advance and for large number of nodes visited. Forwarding is suitable for a small number of migrations; it is not appropriate for long chains.

In many cases, locating agents is application specific. Even if the agent is successfully located, it might migrate further away by the time the location is reported back to the requesting node and communication or delivery of a control message is attempted. This is especially critical in cases of control messages such as *meet*, *suspend* or *kill*. In such cases it is required to perform optimizations, such as to batch a locating request with the control message. This eliminates the delay between the time the agent is located and the control message is delivered to the visiting agent system. This may not be sufficient for highly dynamic agent applications or heavy loaded nodes in the case of forwarding locating strategy. Instead, the updating and/or registering strategy needs to be used, combined with trapping the agent when registering/updating its location. This way agent movement is delayed until communication/control messages are delivered.

4.5 Mobility

When an agent migrates, its state is extracted from the source agent system and transferred to its destination agent system where it is restored into a new instance of an agent object. During transfer, only site independent information is transferred. In the case of communication channels, this information consists of the agent names that the migrating agent had opened channels with as well as their current location. The state relevant to each particular node is transient, i.e., it is discarded. For example the sockets maintained in the agent control object are closed and then reopened in the remote agent control object. Figure 10 describes the agent state that is transferred. The state extraction starts at the application level, where the application state is serialized (non-transient data), then the state of the agent control is serialized (agent resources, such as agent limits and logging data). This state is then transferred to the remote node through the cooperation of Mover objects in the source and destination agent system. The Mover objects involve negotiation based on the agentPolicy and the destination node hostPolicy.

Mobility is based on messaging, where the message object is the bucket containing the agent and related resources. When an agent arrives at a node, the Mover creates a new instance of the AC object (unless there already is one for that agent - in case of the agent is returning to an AE which it left earlier. All other agent related objects are instantiated from the serialized versions in the bucket. Objects are loaded using the class loader associated with the bucket.

We do not provide for sharing of objects remotely, i.e. as an agent migrates to another node, it should not maintain any references to object on the source node. Our experience is that distributed shared state is very hard to support at the system level, it is more appropriate to rely on distributed shared memory packages for such needs [4].

4.6 Resource Management

One of the initial goals of the MOA project was to support extensive resource control of various MOA resources. The following limits are enforced on MOA resources:

- agent: lifetime, places, hops, open channels, clones
- place: lifetime, nested places, open channels, agents
- agent environment: agents, places, channels

These limits are verified upon each MOA function that can impact the values, such as move, open channel, etc. Should the limits be exceeded, the function is interrupted and the appropriate exception is thrown to the component that invoked the function.

Prior to being accepted at a node, the agent negotiates which and how many MOA resources it can utilize at the visiting MOA system.This is achieved by calculating local policy from the agent policy and host policy. The agent local policy is enforced during its lifetime at the visiting MOA system.

We did not address limitations on resources that are not supported by JVM, such as size of VM, amount of processing, size of files used, etc. Whereas it would be possible to enforce some of them by making modifications to the JVM, we refrained from any deviation from *de facto* standard solutions. Imposing resource limits has impacted the design and implementation of the MOA system. This is especially true for communication component.

4.7 Security

The first MOA release is fully compatible with the JDK 1.1 security manager; however, no security manager has actually been implemented. Many security features were left open for the next release, such as the work on authentication, and authorization of agents. We have actually implemented only the following features.

Thread switching was employed to allow conformance with the Java security model. Services are provided by threads containing only trusted classes. When a MOA system thread has to switch the trust boundaries (for example in case of an incoming message, or a request for opening a channel), the system thread is instead suspended. The request is passed to a queue, which is serviced by a pool of threads allocated for that specific trust domain. A thread from this pool processes this request by calling the application specific methods. The suspended system thread resumes either upon receipt of the response from application, or upon the timeout, whichever happens first. This way, the application is prevented from stalling the system by thread exhaustion, or by impacting performance through overusing system threads. In addition, resource usage is tracked on a per sandbox basis.

Each agent has its own name space as defined by the bucket in which it is transported. A name space consists primarily of bytecodes and serialized objects. One complication arises when an agent returns to a place that it had left. In this case, the name space is a combination of the original bytecodes and the returning objects. This is achieved by nesting the returning bucket (with the meaning of classloader in this context) within the bucket of the remaining place.

We are using the standard JAR file format for passing agents. This format has provision for digital signatures, allowing for authentication. However, we have not addressed authentication at the moment. It is left as an open issue, even though we have considered its deployment during design and implementation. For example, the agent's authenticity will be maintained as a part of the agent's name.

4.8 User Interface

MOA's User Interface provides users with various types of interactive monitoring and debugging services. An applet-based and a script-based User Interface are provided. The script based interface is primarily used for testing purposes whereas the applet-based interface is used by the user to interact with the MOA system.

The User Interface is a component in the front end of the system (see Figure 1). It supports interaction with both the home and remote system. A user can log into the system from any remote location. The login information is verified at the home agent system and a list of all the agents (preconfigured applications available for launching and already launched applications) is returned to the user as a result of a successful login. The user can select any preconfigured application agent and launch it to a remote or local destination.

At any point of time, the user can launch a preconfigured agent, send a command to suspend or kill a selected agent and monitor agent-related activities. An agent can be queried by specifying its start time, duration (to determine which log records to access) and a query pattern. The home of an agent maintains a cyclic array of agent snapshots (captures of agent's state at different times). The user can fetch a snapshot and use it to start a new application agent. The interface accepts various types of messages pertaining to an agent's movement, notices, system statistics and log items. All these messages are displayable.

4.9 MOA Tools

We also developed the tools needed for manipulation of Java Beans. Even though many new tools are becoming available commercially, or will be developed soon, we needed some functionality that was not available at the time of development. In particular we developed MoaBatch program for instantiating Beans and MOAJar for manipulating JAR files.

MoaBatch is a simple script program (726 lines of source) which lets you instantiate beans (saving them to disk as serialized objects) and edit the properties of these beans. It can not use the property editors which do not support text. Even so, MoaBatch works with all of the property editors provided with the BDK except for the font editor. MoaBatch fully supports indexed properties. Source and executables are available from http://www.camb.opengroup.org/~laforge/java/moabatch/. Some of the commands that MoaBatch supports for manipulation of beans are included below:

- **Instantiate X** - create a bean X.ser
- **Properties X** - list the properties of bean X
- **Limit N -** limit the display of elements of an indexed property
- **Set X Y Z** - set property Y of bean X to Z
- **SetAt X Y I Z -** set property Y at I of bean X to Z

MOAJar is a GUI utility for editing JAR files. It is layered on top of an API for manipulating the JAR file contents and manifest.The MoaJar program supports:

- Add, remove, extract, or rename a file in the JAR.
- Edit the name/value attributes in the JAR manifest.
- Create a serialized object from a class in JAR or on CLASSPATH.
- Edit the properties of a serialized object in the JAR.

4.10 Standardization

During the development of MOA, we participated at the OMG Mobile Agents Facility (MAF) standard [28]. MAF represents an attempt by General Magic, IBM, Crystaliz, GMD and the Open Group to establish a standard for mobile agents using CORBA. It standardizes agent control, locating, and migration. It does not address communication among the agents. This participation was intertwined with the development of MOA. For example, our experience with MOA has impacted some of the choices in MAF, and conversely, some of MAF specification choices have impacted MOA. In particular, our experience with locating contributed to standardization. MAF impacted our selection of interfaces for the name server, as well as for the naming in future versions of MOA. OMG MAF is important for enlarging the base of agent systems that can accept visiting agents.

5 MOA Current Status

The MOA project started in the summer of 1996. It had four people on staff on average (it spanned from two people up to six people over the life of project). It will be delivered by the end of 1997. It was developed mainly on Windows NT, PC-based machines. We were mainly using the bare JDK for development, although throughout various phases of project and for various purposes, developers have also been using other tools, such as Symantec Cafe, J++ and Java Studio. The main reason for using the raw JDK was due to relatively slow response of industry to Java Beans development.

For development, we have used ClearCase of ATRIA. While we highly regard this tool in general and especially on UNIX machines (we had other development on HP/UX), the match between Clearcase and the JDK on NT was not a good experience. The problems consisted of the use of upper case letters for file names, very poor response time and interference with debugging. By the very end, we had found ways around all these problems, but they mostly consisted of *ad hoc* solutions, and by staying away from ClearCase as much as possible. The bare minimum consisted of maintaining the source code control, and Clearcase was indeed very suitable for this.

At the moment of writing this paper, the MOA system consists of approximately 27000 lines of code (including comments), organized in 16 packages, 132 classes, and 10 interfaces. This does not include test programs, which we developed as unit tests for most packages, and 25 test scenarios which exercised various aspects of the system. The tests only (including the configuration files, which represent the smaller percent) represent an additional 9000 lines of code. Despite relatively significant effort invested in different type of testing (and we are still in the testing phase of the project), MOA still requires a lot of testing for robustness and performance improvement, in order to productize the project. The footprint (accumulative size of classes) of the whole MOA system, along with the test programs, is approximately 730KB.

6 Applications

"Rent-A-Soft**"** is a demo program, presented at Uniforum. The idea is to use agents to help out with distributing and renting software packages. This is applicable for relatively expensive packages, or for cheaper software (such as games) assuming large quantities. A chain of participants is envisioned, such as producer, wholesale renter, renting departments, the company which rents and its own departments which sub-rent and occasionally exchange software. By encapsulating information about the renting source, duration and usage of the package, an otherwise complex inventory tracking process is replaced. Each encapsulated layer can associate an agent responsible for specific activities, such as revoking the rent, statistics maintenance, etc. Communication in presence of mobility can be used to revoke the rent within the same security domains. For example, the company which rents may revoke certain copies of package within its own departments, it can install new versions of software, and dynam-

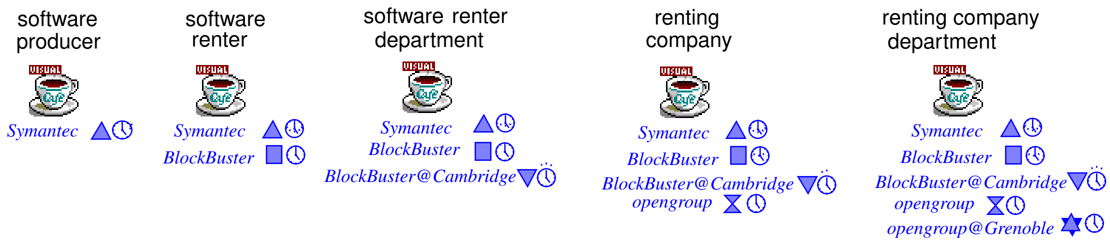

Figure 11: Rent-a-soft: *each level (producer, renter, renting company, and their departments) encapsulates its data and supporting code for maintaining the renting process, upgrades, inventory, etc.*

ically monitor software use. The Rent-a-soft application is presented in Figure 11.

Radio Communication. The Grenoble facility of the Open Group Research Institute is planning to use the MOA project for a police force application over radio connections [16]. This is a type of application that is very nicely suited for mobile agents. Radio communication has slow and unreliable links, allowing mobile agents to exploit the locality of reference. Another important feature that prevents alternative solutions based on the client server model to be used as effectively is the unpredictable availability of connections. It can easily happen that connections with the centralized server are broken (e.g. a police car enters a tunnel). In such a case, mobile agents can still continue to be functional, e.g. by cooperating with, or visiting other available locations (e.g. another police car that is also in the tunnel).

Security. Even though mobile agents are considered to suffer from (still) inadequate security, they can help to solve some security problems. In certain cases it is not allowed to give access to the actual data stored in a security domain, whereas it is allowed to provide some attributes of the data or some other information about the data. For example, Swiss banks do not allow WWW access to its old accounts since the World War II, but they allow anyone checking if the user (or someone from his family, based on the last name) had an account opened. This type of application is a relatively simple query. Imagine a more complex application, where queries need to do complex, arbitrary searches across the whole security domain. Such an application can be easily achieved using mobile agents. A mobile agent would be allowed to enter the security domain. While in the security domain, it can do activities within certain limitations. Before leaving the security domain, it can be inspected to see what data it takes out; alternatively only certain data could be allowed to leave. There are opportunities establishing covert channels, but depending on how secret the data is and how much information is allowed to leave, this solution could be acceptable in a range of applications. The Open Group RI has an ongoing proposal for the use of mobile agents for improving security.

7 Lessons Learned

In this section we try to summarize various lessons we learned while developing MOA.

Operating System Support vs. Middleware. There is significantly reduced work in the development of operating systems. The current trend is towards using NT on lower end systems and some version of UNIX on high end servers. Linux is a dominating freely available version of an OS, and there are also a few real-time executives. In many cases, operating system modules are being replaced with middleware solutions. Similar is true for the MOA project.

Nevertheless, many operating systems techniques can be applied in the development of middleware systems. Again, the same applies in the development of MOA. We have drawn substantial experience from the area of operating systems, such as communication channels and messaging protocols; locating and naming of mobile agents; resource management; negotiation policies, synchronization among agents, etc.

Transparency in communication (maintaining channels across migration) was more complex to support than we originally thought. We were aware that this is a hard goal to achieve, but we were hoping that relaxing assumptions we made would make it simpler to implement.

Resource management was relatively straightforward to design and implement. We believe that its extensive usage will even more demonstrate its ultimate benefits. We strongly recommend that resource management be initially planned for development of agent systems. We shall heavily rely on it for some of the future work related to policies for management of agent based systems.

Components. Component based computing has somewhat slowed us down during the development. Compliance with component model does not come for free, there are costs both in terms of development effort, as well as the run time costs. The learning curve was high to get accustomed to Java Beans; we had to provide additional methods required to inspect/set properties; we had to take care that all classes are serializable; we had to create jar files for both the agent application and system; it is required to link (or wire) components once they are loaded. Nevertheless, we feel that the benefits have at least returned the investment so far, and we believe that the benefits will significantly outweigh the investment once we start using and especially configuring a MOA system and MOA applications.

Immediate benefits of complying with the component model compatibility were stronger enforcement of component boundaries than is the case with object boundaries (components are loaded instead of constructed). In addition, component boundaries enforced careful design of what is serialized, particularly useful for application development

In the future we expect even higher benefits of component model compatibility, allowing for inspection of visiting agents, reconfiguring agent applications, as well as reconfiguring agent systems. Evolvement of the MOA system will be easier, since all changes and enhancements will be isolated to a single component.

Finally, it is too early elaborate on the benefits of participating in the **OMG MAF** proposal. Certainly it was a useful experience to collaborate with other implementors of mobile agent systems. It was clear that we were solving similar problems, sometimes finding different solutions. Because of the different underlying infrastructure, the current compliance is still a future goal, because we need to come up with a reference implementation first. At the moment, we have taken care that nothing stands in the way in the terms of the MOA design to prevent us from switching to different communication infrastructure.

8 Related Work

There are three classes of related work to MOA. The first class consists of process migration, the second of distributed systems on the Web and the third of mobile agents.

Charlotte process migration [3] extensively dealt with the interprocess communication among the migrating processes. Artsy et al. supported a forwarding scheme by leaving trails behind migrating processes. Process migration in the **Sprite** operating system supported the notion of a home node [12]. The home node has the ultimate knowledge of the migrated away process. In the **V Kernel** process migration [33], migrating processes are located by searching them. **Emerald** supports fine grain mobility on a small scale network (up to a hundred nodes) addressing mobility at the language level; an object, or graphs of objects are moved among nodes [20]. In Mach task migration, transparency of communication and resource maintenance is achieved at the microkernel level [24]. A comprehensive survey of process migration is available at [23]. A theoretical description of mobility in form of Actors is presented in [1].

Two distributed object based systems, on the Web scale, explore similar issues as MOA. **Legion** is an object-based, meta-systems software project, developed at University of Virginia [14]. It attempts to provide a single, coherent virtual machine that addresses the issues of scalability, programming ease, fault tolerance, site autonomicity, security for users and resource providers. **Globe** is an object-based wide-area distributed system constructed as a middleware layer on top of existing networks and operating systems [18]. It is based on the concept of a distributed shared object whose state can be physically distributed across multiple machines at the same time. Each object fully encapsulates implementation aspects, including communication protocols, replication strategies, and distribution and migration of state.

Telescript is the first commercial implementation of the mobile agent concept [36]. Recently, it was discontinued and reimplemented in Java, under the name Odyssey. **AgentTcl** is a mobile agent system implemented in the Tcl language [21]. It has two components: a special Tcl interpreter that executes the Tcl agents, and a server that runs on each machine to which agents can be sent. It uses the SafeTcl model for security. **Aglets** is one of the first mobile agent systems written in Java [2]. It supports rich communication semantics (location independency, synchronous, asynchronous and multicast). **Mole** project at University of Stuttgart was one of the first academic efforts in mobile agents in Java [26]. It collaborates with a few industrial partners, such as Siemens and Tandem. **Concordia** supports agent persistence and recovery [9]. Collaborative work is based on event manager and two forms of asynchronous distributed events: selected and group-oriented. **Ara** is a Java-based agent system that applied some changes to the JVM in exchange for increased functionality, such as maintaining thread execution context and imposing limits on memory usage [29]. **Tacoma** and its descendent T2 address fault tolerance and security issues [19]. **Voyager** is a Java-based system for developing distributed applications using mobile objects and agents. It includes an ORB with support for migration, services for persistence, scalable group communication, and basic directory services.

Of the systems presented, most elaborate schemes for maintaining communication channels across migration were implemented in process migration. This was achieved at the cost of complexity introduced in the operating system [12, 24, 25]. Voyager also supports communication with the migrated away agent, but it relies only on the forwarding strategy. Even though this strategy may appeal superior to others (see Section 4.4), it is really the combination of different strategies that offers most benefits to an application writer.

Almost all the systems described provide some support for resource management. None of them, to our knowledge, have made an elaborate use of the resource information to pursue negotiation and control.

None of the systems that we described is compliant with the component model. MOA was developed later than most of the agent systems, allowing it to timely overlap with the development of Java Beans. This is one of the rare case when being late happened to be an advantage.

Of the agent systems we described, Aglets is the only other agent system that plans to pursue a MAF reference implementation. MOA and Aglets are currently similar with regards to MAF compliance, i.e. both are Java-, rather than CORBA-oriented. It will be required to adapt security and communication models to adjust to MAF requirements.

9 Conclusion and Future Work

In this paper we described the design and implementation of the MOA project. In particular we presented the MOA object model, component model and described particular components, such as communication, naming and locating, mobility, and resource management. We also drew some lessons learned during development. MOA contributions consist of: supporting agent collaboration by maintaining communication channels across migration; providing basic support for denial of service attacks by extensive resource management and negotiation policies; compliance with the Java beans component model, leading to better configurability; and complying with the OMG MAF standard.

There are many mobile agent systems available nowadays, both from academia and from industry. Even though MOA represents yet another new mobile object system among many research vehicles today, we believe that we have distinguished it sufficiently enough to justify its development. In particular, we believe that it was easier to achieve compliance with the component model while designing the system; similar reasoning applies to managing resources, and to maintaining communication channels.

The lessons we learned consist of resemblance of middleware solutions to experience with operating systems. Our experience with the component model distinguishes between disadvantages and benefits of complying with the component model. We strongly believe that latter will outweigh the former already for moderate requirements for configurability. We introduced a lot of complexity by maintaining communication transparency. Resource maintenance proved to be very useful with expectations to significant benefits in the future. Finally, OMG MAF standard impacted only the design decisions of MOA so far.

The future work consists of four areas. First, we plan extensively to improve security. In particular, we plan to include authentication, authorization (as well as delegation of agent credentials), integrity checking, and the trust model of MOA. The second area consists of applications. We plan to support a number of new applications. The third area addresses improvements to the current implementation, in particular related to performance and robustness. Finally, we plan to demonstrate interoperability in practice, by interoperating with another OMG MAF reference platform, such as Aglets.

Acknowledgments

We are grateful to Shai Guday and Holger Peine for reviewing this paper. They significantly improved its presentation and contents. Rosemary Hudson undertook impossible task to insert all missing articles and eliminating the superfluous ones that an author, a non-native English speaker, introduced.

References

[1] Agha, G., “A Model of Concurrent Computation in Distributed Systems”, *MIT Press,* Cambridge, MA, 1987.

[2] Aglets, http://www.ibm.co.jp/trl/projects/aglets/

[3] Artsy, Y. and Finkel, R. (September 1989). Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, pages 47–56.

[4] Black, D., Milojicic, D., Langerman, A., Dominijanni, M., Dean, R., Sears, S., “Distributed Memory Management”, accepted for publication, Software Practice & Experience, 1997.

[5] Bradshaw, J., “Software Agents”, *AAAI/MIT Press,* 1996.

[6] Brookshire, D., “Java Beans, Developer’s Reference”, *New Riders Publishing,* Indianopolis, Indiana, 1997.

[7] Chess, D., Grossof B., Harrison, C., Levine, D., Parris, C., Tsudik, G., “Itinerant Agents for Mobile Computing”, *IEEE Personal Communications Magazine*, October 1995.

[8] Cockayne, W., and Zyda, M., “Mobile Agents: Explanations and Examples”, *Manning*, 1997.

[9] Concordia: “Concordia: An Infrastructure for Collaborating Mobile Agents” *Proc. of Workshop on Mobile Agents MA’97*, Berlin, April 7-8th. LNCS 1219, Springer Verlag.

[10] Cybenko, G., Spontaneous comment during Transportable Agents Workshop, September 1997.

[11] DARPA Broad Agency Announcement, 98-01, “Agent-Based Systems”, http://ballston.prc.com/baa9801/abspipv1.htm

[12] Douglis, F. and Ousterhout, J. (August 1991). Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software-Practice and Experience*, 21(8):757–785.

[13] Ford, W., Baum, M., “Secure Electronic Commerce”, *Prentice Hall*, New Jersey, 1997.

[14] Grimshaw, A., et al. “The Legion Vision of a Worldwide Virtual Computer”, CACM, vol. 40, no 1, Jan. 1997, pp 39-45.

[15] Goldszmidt, G., Yemini, Y., “Distributed Management by Delegating Mobile Agents”, Proc. of the 15th ICDCS, Vancouver, British Columbia, June 1995.

[16] Grenoble Open Group Research Institute, ANIMA project, http://www.gr.opengroup.org/anima/.

[17] Guideware Corporation, http://www.guideware.com.

[18] Homburg, P., van Steen, M., and Tanenbaum A., “An Architecture for A Wide Area Distributed System”, Proc. of the Seventh SIGOPS European Workshop, Connemara Ireland, September 1986, pp 75-82.

[19] Johansen, D., van Renesse, R., and Schneider, F., “Operating system support for mobile agents”, *Proc. of the 5th. IEEE HOTOS Workshop*, Orcas Island, USA (4th-5th May, 1995),.

[20] Jul, E., Levy, H., Hutchinson, N., Black, A., “Fine-Grained Mobility in the Emerald System”, *ACM Transactions on Computer Systems*, vol 6, no 1, February 1988, pp 109-133.

[21] Kotz, D., et al., “Mobile Agents for Mobile Internet Computing”, July/August 1997, IEEE Internet Computing, vol 1, no 4, pp 58-67.

[22] Lange, D., Oshima, M., “Java Agent API: Programming and Deploying Aglets with Java”, Addison Wesley, expected publication date, Winter 1998.

[23] Milojicic, D., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S., “Process Migration Survey”, *The Open Group Research Institute, Collected Papers, vol. 5, March 1997*.

[24] Milojicic, D., Zint, W., Dangel, A., Giese, P., “Task Migration on the top of the Mach Microkernel”, Proceedings of the third USENIX Mach Symposium, April 1993, 273-290, Santa Fe, New Mexico.

[25] Milojicic, D., Douglis, F., Wheeler, R., Guday, S., “Mobility - Migration, Mobile Computing and Agents”, vol I (Selected Papers) and vol II (Textbook), Addison Wesley, expected dates of publication, Fall 1998 and 1999 respectively.

[26] Mole - A Java Base Mobile Agent System”, in *Special Issues in OO Programming*, pp 327-334, Workshop Reader of the 10th European Conf on OO Programming, 1996, http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.htm

[27] Odyssey Web Page, http://www.genmagic.com/agents/odyssey.html.

[28] OMG Mobile Agent Facility Interoperability Facilities Specification (MAF), OMG TC Document ORBOS/97-10-05, also available from http://www.opengroup.org/~dejan/maf/draft10.

[29] Peine, H., and Stolpmann, T., “The Architecture of the Ara Platform for Mobile Agents”, *Proc of the First Intl Workshop on Mobile Agents MA’97, Berlin,* April 7-8. LNCS 1219, Springer Verlag, http://www.uni-kl.de/AG-Nehmer/Ara

[30] Riggs, R., et al.,”Pickling State in the Java System,” *Proc. of the USENIX 1996 Conference on Object-Oriented Technologies (COOTS)*, pp 241-250.

[31] Ranganathan, M., Acharya, A., Sharma, S., Saltz. J.,., “Network-aware Mobile Programs”, *Proceedings of the Annual Usenix 1997 Conf.*, January 6-10, Anaheim, California, USA.

[32] Shoch, J., Hupp., J., “The Worms Programs - Early Experience with Distributed Computing”, Communications of the ACM, 25 (3), pp 172-180, March 1982.

[33] Theimer, M., Lantz, K., and Cheriton, D. (December 1985). Preemptable Remote Execution Facilities for the V System. *Proc. of the 10th ACM Symp. on OS Principles*, pages 2–12.

[34] Vitek, J., and Tschudin, C., “Mobile Objects Systems: Towards the Programmable Internet”, *Springer Verlag,* April 1997.

[35] Voyager, ObjectSpace, http://www.objectspace.com/voyager.

[36] White, J., “Telescript Technology: Mobile Agents”, General Magic White Paper (http://www.genmagic.com/Telescript/Whitepapers/wp4/whitepaper-4.html).

[37] White, J., et al., “System and Method for Distributed Computation Based upon the Movement, Execution, and Interaction of Processes in a Network”, *United States Patent,* no 5603031, February 1997.

[38] Wollrath, A., et al., “A Distributed Object Model for the Java System,” *Proc. USENIX 1996 Conf. on Object-Oriented Technologies (COOTS)*, pp. 219-231.