Mobile Objects and Agents (MOA)

Functional Specification

Dejan Milojicic, William LaForge and Deepika Chauhan

The Open Group, Research Institute
11 Cambridge Center
Cambridge, MA 02142
[d.milojicic, d.chauhan, b.laforge]@opengroup.org

Version 1.2.1

1 Introduction

The Functional Specification is the second document related to the Mobile Objects and Agents (MOA) project. It is preceded by the Architectural Specification, and followed by the Design Specification.

This document should be read in conjunction with the APIs generated from the source code by javadoc, reflect currently implemented interfaces. The functional specification provides higher level description, and also reflects logical relationship between user level abstractions and APIs, whereas in APIs some methods and state reside in internal classes.

1.1 Purpose of the Document

The purpose of this document is to describe API for the MOA project and discuss important concepts in MOA such as naming, addressing and locating of agents and other abstractions.

1.2 Scope of the System

Although the agents can be used in different settings, our ultimate target is to deploy them on the Whole Web. Another important setting will be intranets for which this particular project may be even more suited due to the temporary security limitations.

1.3 Definitions, Acronyms and Abbreviations

Mobile objects are programmed entities that can roam a network and contain their own threads of control. Mobile objects are called “agents” if they act on behalf of a user or another object. They maintain state across migration and make decisions using it. They differ from downloadable applets in that they independently roam among different machines on a network. Mobile objects and agents evoke process migration in traditional distributed systems. However, they have significant advantages and pose different challenges. Mobile agents have been a topic of research in artificial intelligence and office information systems for over a decade, but it is only recent emergence of the Web and widespread availability of safe languages, that makes their deployment both practical and attractive.

Some of the frequently used acronyms in this specification are:

MOA - Mobile Objects and Agents,

URL - Uniform Resource Locator, the way how to address resources on the Web; and

URN - Uniform Resource Names, the proposed scheme how to name the resources on the web.

TBD - stands for "To Be Determined" and it will denote those parts of the document that will be determined at some later point of time.

JDK - stands for the Java Development Kit.

AE - stands for the Agent Environment, which is an abstraction (and also class) agents visit on various nodes.

1.4 References

This document expands on the preceding document:

> Milojicic, D. and Guo, K., "Mobile Agents, Architectural Specification", The Open Group Research Institute, available off of the MOA Web page: http://www.osf.org/RI/DMO/dmo.htm (All subsequent documents will be available from the same location).

The development will be pursued using Java, and therefore we shall exploit a lot of Java literature, such as:

> Arnold, K., Gosling, J., *"The Java Programming Language"*, Addison Wesley, 1996.
>
> Gosling, J., Joy, B., Steele, G., *"The Java Language Specification"*, Addison Wesley, 1996.
>
> Gosling, J., Yellin., F., *"The Java Application Programming Interface, Volume 1, Core Packages"*, Addison Wesley, 1996.
>
> Gosling, J., Yellin., F., *"The Java Application Programming Interface, Volume 1, Window Toolkit and Applets"*, Addison Wesley, 1996.

The following two technical reports can serve as an additional source of information on mobile agents or their predecessors, process migration.

> Milojicic, D, Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S., "Process Migration Survey", available from http://www.osf.org/~dejan/papers/indx.htm.

Milojicic, D, Bolinger, D., Zurko, M.E., Mazer, M., “Mobile Objects and Agents”, available from http://www.osf.org/~dejan/papers/moa.htm.

1.5 Overview of the document

Section 2 provides a general description of the project. It describes: product perspective, functional requirements, system properties, compatibility with other systems, system constraints, user characteristics, operational environment, goals and non-goals, assumptions and dependencies. Section 3 addresses specific requirements of mobile agents. In particular, we address naming, interfaces, system properties, constraints and development requirements. Section 4 deals with deferred features and explicit restrictions of the project. Section 5 addresses quality and assurance.

2 General Description

2.1 Product perspective

Mobile agents are a relatively new paradigm on the Web, however, they have a long history in related areas of process migration in the field of operating systems and software agents in the field of artificial intelligence. While both predecessors have reputation of “mechanisms searching for the problem”, the prospects for mobile agents, at this moment seem to be quite encouraging. There are already a number of companies providing agents as products, such as, General Magic and FTP Software. A couple of standardization processes are in progress (FIPA, OMG), and a number of research projects and both industry and academia are underway.

2.2 Overview of functional requirements

2.2.1 Naming, Addressing and Locating

Naming is a way of distinguishing who/what an object is. An object can have multiple names, or a name can refer to multiple objects. **Addressing** is a way of specifying location of an object. Using an address, an object can be addressed, i.e. communicated with. In our object model, agents and places are named and addressable (more will be explained later) Since an agent is a mobile object it is also required to locate the current location of an agent. **Locating** is a way of finding the current address (location) of a mobile object (agent). Naming and addressing will be covered in this document. Locating is not exposed to the user, and as such it is covered to more extent in the design document.

For naming and addressing of agents to be effective, there are a few criteria it should fulfill:

- The names should be unique, no two agents, or two places in the WWW should have the same name and address at the same time.
- The naming should be comprehensible to users and easy to use by programmers.
- The naming should conform to standard naming currently used in the community.

- The naming should be extensible.
- Location transparency is not a prerequisite, but it can simplify agent manipulation
- The system should scale well enough with the number of agents and nodes they can visit.
- There should be an ultimate naming authority that can resolve the agent name if all other naming attempts failed.

In the Section 3.1 we revisit these characteristics to investigate how the proposed scheme matches the requirements.

While proposing naming and addressing of an agent system, the following assumptions: can be made about the objects:

- Mobile agents are typically short lived - a couple of weeks at most. Therefore, the names may be recycled; the age and time-to-live attributes are transferred along with the agent when the agent is moved. The time to live, enhanced with a certain guarantee factor, secures that all existing references to an agent that died are recycled, thereby guaranteeing the uniqueness of agent names.
- There is a relationship between the user, original agent created by that user and the cloned agents originating from a particular agent. Therefore, it would be useful to maintain the association in the naming scheme.
- Agents resemble processes as opposed to files, and hence the parent/child (family/generation) model is more appropriate than directory/file type of naming, whereas the latter is more appropriate for the naming of places.
- Agents can have arbitrary frequency of movement, depending on the type of application.

2.2.2 Communication

The communication endpoints can be either established among places, or among agents, or between places and agents. From the user perspective, nevertheless, it is an agent that communication is being directed to. However, in certain cases, being able to send and receive from the remained places allows additional functionality. Whenever a channel is opened, a separate thread is associated with it, thereby making a connection "active". Having the ability to have open channels either to places or to agents allows to two different kinds of functionality:

- Collaborative agents can maintain open communication channels between them across migrations (channels between agents).
- Sending data to a certain place where the owner can later inspect it.

2.2.3 Resource Management

Although Java environment maintains its own notion of resource management, the agent system imposes additional restrictions to the resources agents can consume. In other words, proper agent systems should be self-limiting, good citizens of the Internet. Therefore limitations are imposed on various resources an agent can consume on a node, e.g., the number of clones, the time to live, the number of places it can create, etc. Some of these assumptions help in garbage collecting, for example, when time expires the agent as well as places and other resources owned by the agent are freed.

2.3 Overview of system properties

The system has to be fast and small, to the extent that the current Java technology allows. Specifically, we may incur some performance problems with Java interpreter in the beginning, but it is expected that this will vanish with the introduction of Just-in-Time (JIT) compilers and standard Java compilers (where downloading is not an issue).

2.4 Compatibility

The project assumes compatibility with Java standards. The project uses Java language and utilities, such as object serialization.

In addition, it adheres to some accepted terminology in the agent world (for example the concepts of Agents and Places introduced by Telescript).

2.5 Overview of constraints

There exist certain security constraints imposed by the current Web browsers. These constraints should be relaxed as security mechanisms are enhanced and deployed on the Web and supported by Java. For example, it is impossible to communicate from the downloaded applet with any other node than one the applet was downloaded from. Similarly, downloaded applets are not allowed to access local files. Therefore, it is not possible to use agents within applets for the time being. We expect this to change in the near future.

2.6 User characteristics

User friendly interface for controlling and manipulating agents, e.g: graphical interface for starting, killing, stopping agents; graphic interface for tracking agents (relatively simple graphics, visualization can be project per se); user/programer customized names (internally, longer, complete names might be supported, but the user can use short names).

Simple solutions, with convenient analogy to real-word examples (e.g. agents, places, tickets, etc.).

2.7 Operational environment

The user will operate through graphical interface. Through the graphic interface users can access the, agent environment and agent services that run on top of the Java virtual machine. Agents may contact specific services provided by dedicated servers (not part of the project), for example JDBC.

2.8 Goals and Non-Goals

The following goals are envisioned:

- Provide a working and usable agent environment.
- Sufficient performance to warrant deployment in real-world problems.

Non-goals in the near term are:

- As mentioned in earlier documents, security will be considered in the design and implementation, but there will not be any separate work on security.
- There are no requirements at the moment to support persistence of agents, i.e. to store them at the persistent storage and then restart them. This should be relatively easy to support, but there is neither requirement nor commitment to do so and it will be pursued subject to time and personnel availability. Nevertheless, the required mechanisms for achieving this, such as support for making *snapshots*, will be provided. It is the matter of higher level policy how to make use of this, and when to restart the agents.
- There will be no attempt to support fault resilience of agents. For example to use persistence in order to store agent snapshots at intermediate host and then to recreate them in case an agent died due to host or network failure.

2.9 Assumptions and dependencies

This project entirely depends on Java developments. There is an assumption that Java support will be ubiquitous in the near future. We also depend on other Java related support, such as object serialization, The alternatives, such as Active X or Corba, although quite attractive, were not considered because of the time constraints.

Finally, assumption is that we should start from a simple agent model, which would allow us later to adapt to whichever standard is adopted (if any at all) in the future (e.g. Aglets)

3 Specific Requirements

This section analyzes requirements in naming and addressing, and then presents the APIs for the MOA project.

3.1 Naming and Addressing

This subsection describes naming, addressing and locating of agents and other abstractions in MOA. Although not the explicit part of API, this aspect has the major impact on the whole API. Therefore, it is analyzed and described in more detail than normally required in a functional specification. Naming and addressing are modeled after the general Internet naming (URLs [URI94], [URL94]), enhanced to support more dynamic needs required by agents, possibly integrated within URNs [URN94], once they become standard). MOA uses a subset of the URL, i.e., host and port number, as this is more appropriate in naming of agents and places.

3.1.1 Naming of Agent Systems

Agent Systems are named and addressed by the *host:port* tuple. This is a reasonable choice at present as URLs are more heavyweight in nature than what is required in the scope of this project. The agent system name is the starting point of the naming scheme for other MOA abstractions such as agents and places.

3.1.2 Naming of Agents

An agent name will uniquely name and address an agent system-wide, i.e., across all agent systems across all nodes. The name is of the form *h:p#f_l.g*, where

h:p is the home agent system name

f is the agent family name

l is the launch number

g is the generation number, e.g., 1.2.3.1.2

3.1.3 Naming of Original Agents

Original agents are created as fresh applications, as opposed to being cloned from an existing agent. The agent name of an original agent is considered as a family name for all agents cloned from the original agent.

The agents are named relative to the agent system in which they were originally created. The agent name is extended with the user specified extension, e.g. “agent”, provided by the agent application. If application does not provide it, the default name is provided by the agent system.

When addressed within an agent system, or by the owner, only agent extension name is used instead of the full name including the agent system name. When the name is passed to another agent system, the full name is used.

The agent name is represented by the following string:

host:port#f_l, where f, l are defined in Section 3.1.2

If another agent is created with the same name (either because the same or another user has specified so, or because the agent system notices that the same class would be created), then an extension is added to the same name with the appropriate number, in an increasing order. Since agents have short life cycle (a couple of weeks at most, for event-based agents that wait for some event to happen), the names are recycled.

3.1.4 Naming of Cloned agents

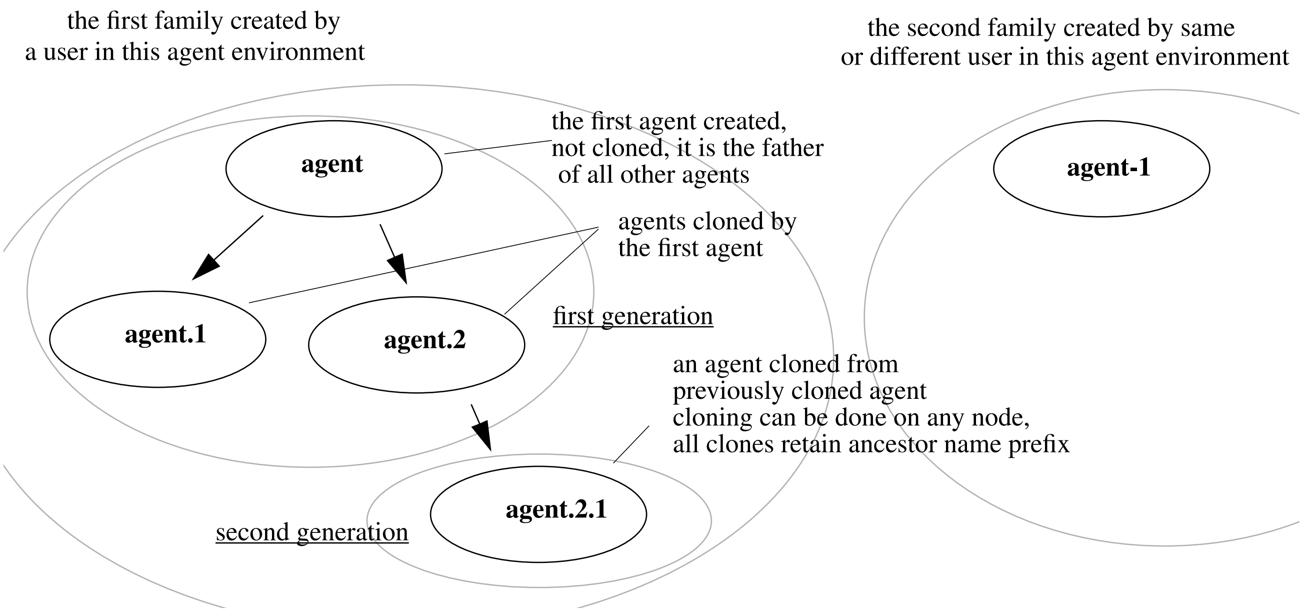
If an agent is cloned, it gets a new generation number represented by a delimiter “.” and an attached number. The resulting name hierarchy is presented in Figure 1 and it can be formally described by:

host:port#f_l.g, where f, l and g are defined in Section 3.1.2

Names are recycled either after it is determined that an agent has been terminated, or after its lifetime has expired (even though nothing is known about his destiny). This can not result in two agents existing with the same name at the same time.

Aliases could be introduced as a utility for easier naming, so that long names can be avoided. Aliasing would allow names such as “sellingAgent” to be assigned and used instead of the longer version of the name of the same agent. Nevertheless, the full name is required when controlling agents created at another agent environment. Aliasing is left as the future work subject to available time and the real need for it.

FIGURE 1. Hierarchies of Cloned Agents and their Naming



It is also important to note that agents cloned on a node that it is not the home node of their original and ultimate ancestor still have the ancestor's node as a part of their name. A private agent variable contains the information about the location of their birth, but the agent name still indicates the ancestor's home node. Although this might indicate contradiction with the notion of maintaining the agent's birth place as a part of the name, the priority is given to the first agent created, since it is responsible for the existence of this agent (either directly, or indirectly, through its children and grandchildren). Therefore, this name, containing the address of its predecessors can be used for detecting (and potentially killing) the source of all the agents in this family. Furthermore, the fact that an agent was cloned on a certain node is incidental; this information is not important for the application (except maybe for debugging and auditing, purposes).

3.1.5 Naming of Places

Name of a place is issued by the underlying agent system, it is a combination of the agent name, extended with the host and port number of the current node. A place is addressable in the form, such as:

a*gentName% h:p*

Place names can be specified by agent programmer, if not, defaults are provided by the agent system.

3.1.6 Naming and Addressing Requirements Revisited

- The names are unique. No two agents or two places in the WWW have the same name and address at the same time because the agent environments receive unique names upon their creation, and then they proceed creating unique agent and place names.

- The naming is comprehensible to users and it is easy to use by programmers, because names of places are created to correspond to their creation history. In some unusual cases, such as eager creation of cascaded children (the child immediately creates another child), the user specified names can be allocated to prevent the long agent names (such as ‘agent.1.1.1.1.1. . ... 1”).
- The naming is extensible, the design does not make any assumptions on the agent ancestry information embedded into the names (the user can and should use this information, however, to make some knowledgeable decisions based on the agent name), it is just a convenient informative indication for the users of agents, and the actual information is maintained at the source node.
- Location transparency is supported by retaining a unique agent name independently of its movement across the network.
- The system scalability will be addressed through replicated name servers and caching of names on agent environments agents visited recently.
- An ultimate naming authority represented either by the home node, or the alternative node resolves the agent name if all other naming attempts failed.

3.2 Interfaces

The names of variables and their visibility are still not fixed, they are provided in the document for early access and evaluation. The full set of APIs is provided in the html documents generated from the source code by javadoc. This document summarizes fields and methods based on their relationship to certain abstractions, such as agent, place, agent environment, channels, etc., without getting involved into actual implementation, i.e where the interfaces and fields are really implemented (for example, some of these interfaces and fields are really implemented in other classes).

3.2.1 Agent State and Interfaces

Agent State

AgentProperties properties.
Policy agentPolicy.

Agent Interface

```
void move(AEname, boolean);
void meet(AgentName[], Place, long);
Agent clone(int, int, Agent, AEname);
void clone(int, int, Agent);
void log(String, String);
Failure snapShot();
void application(boolean);
AgentChannelAccess openChannel(DestinationName);
void closeOpenChannels();
void receiveIncomingChannel(AgentChannelRequest);
void data(DestinationName, Object);
```

```
Object receiveMessage(DestinationName, Object);
Object sendMessage(DestinationName, Object);
void takedownAC(Failure);
```

__Agent State__

Agent data can be transient, or migrateable. The transient data is maintained only locally to a node, whereas migrateable data persists across migration. The application programmer specifies transient data with the Java "transient" attribute. The internal migrateable data is maintained in specific structures serialized prior to migration. A snapshot of the data can also be taken and transferred back to the home node. Finally, data can also be shared with other agents, if the agents can have reference to each other.

AgentProperties and AgentPolicy are part of the transient agent state and visible to the application. AgentProperties represent characteristics or properties of an agent that are immutable. The constituents of AgentProperties are as follows:

owner - A system dependent variable. It may differ between UNIX, NT, and Window 95 environments, but it is only interpreted at the home node, so it is not of importance on other sites.

familyNumber - This is number associated with every instance of an agent that was created by one or more users at an agent environment. For example, it is "77" for agent hostname:port#77_xx.yy. Family number distinguishes otherwise identical original agents.

generationNumber - this is the agent name extension after the launch number. The generation number denotes the generation to which this particular agent belongs. It can be a string of arbitrary size, for example, "*1.2.1.2.4*", as part of the agent name *agent.1.2.1.2.4*. This particular agent could have been cloned anywhere on the Web, but it is known that it was the fourth child of second child of the first child of the second child of the first instance of the agent named *agent*.

lastChildName - the name of the last child created. The children are given names as an extension of the parent name, so the parent needs to remember the last child's name in order to be able to name the subsequent children and to guarantee the unique names on the Web. Since the agents can create children (clone) on any node, this variable has to be migrateable.

homeAgentEnvironment - designates the URL of the agentEnvironment where agent was created. Note that there can be different agent environments on a node, and agent can visit arbitrary one. From home agent environment, user can find out the home node location.

alternateHomeAgentEnvironment - designates the agent environment that serves as an alternate home agent environment in case that the true one is not present on the network. From alternate home agent environment, user can find out the alternate home node location.

eagerChannelOpen - When an agent moves or attempts to move all channels opened by an agent are closed. When an agent arrives at a node after a successful move the channels are opened and the flag eagerChannelOpen is set to true. When an attempt to move fails the closed channels are re-opened and the eagerChannelOpen flag is set to true.

maxSnapshots - The maximum number of snapshots retained by the home service.

homeLife - Elapsed time in millisecs for which the home service retains the snapshots.

AgentPolicy represents the resource limits of an agent. The agentPolicy is used by the negotiator in determining the policies enforced by the agent system. The following variables are available for resource management purposes:

maxLifeTime - Maximum lifetime of the agent, expressed in seconds. There is a timer associated with each agent, and when it expires it will throw an exception to the agent. This time represents elapsed time, i.e. overall time that an agent can live, not how much CPU time it has used on particular machines. The relative difference in the absolute time on different machines does not impact lifetime accounting, because lifetime is accounted for relative to each node, i.e. the absolute time does not figure in the accounting, only the relative time that an agent spent on a particular node, added to the time it spent on nodes it visited in the past.

maxThreads - Specifies the maximum number of threads that may be requested by an agent.

maxChannels - Specifies the maximum number of channels that may be opened by an agent. This limit is intended to prevent (to some extent) erroneous applications from exhausting the resources of the node.

maxClones - The maximum number of clones an agent can create. The parent can pass a certain number of its own allocated clones to its child. If none is passed, then the child can not invoke the clone method. Clones created by clones consume the parent's quota of allowed clones, *maxClones.*

maxPlaces - maximum number of places an agent may leave behind during the duration of its lifecycle. This is the cumulative number of places that the agent may have left behind along its migration path.

snapshotsRetained - the number of snapshots archived for an agent by the homeservice.

remainingPlaces - the number of places the agent has left behind. It can not leave more places than it is specified by *maxPlaces*.

<u>**Agent Interface**</u>

The API exposed to the agent developer for developing agent applications is as follows.

move*(AEname, boolean)* - Move an agent to a specified agent system. The boolean value indicates if the place will be left behind on completion of the move operation. As a part of the *move* method, negotiation is performed on security and resource restrictions.

meet*(AgentName[]* L*, Place* P*, long* T*)* - Meet the invoking agent at the specified place P for the duration specified T. The inviting agent may invitess more that one agent to the meeting. The invitees to the meeting are specified by the parameter L.

snapshot() - Snapshot the agent relevant state to the home service. This operation will archive the current state of the agent persistently. The archive will permit the application developer to restore the agent from its persistent state.

clone*(int* nn*, int* np, *Agent* C*, AEname* H*)* - Create clone agent(s) of the agent C. The number of clones that may be created nn, cannot exceed *maxClones.* The clones are created at the current agent system or at the agent system H.

clone*(int* nn*, int* np*, Agent* C*)* - the behavior is the same as the clone operation above except that in this case the clones are created at the agent system of the invoking agent.

log*(String* typ*, String* dat*)* - A log of type typ is generated and logged with the logging service. The log information is encoded in the parameter dat.

application*(boolean* ret*)* - This is analogous to a java applet's run() method. The agent application developer overrides this method with program code for the agent application.

openChannel(DestinationName nn) - Opens a communication channel to an agent or a place. The parameter nn specifies the name of the other end of the channel. The channel is a java socket (streams-base) communication channel.

closeOpenChannels() - The current channels that are in an open state are closed.

receiveIncomingChannel(AgentChannelRequest) - This function is overridden by the application developer with application code that handles an incoming communication channel. This function is overridden by the application-level concrete class derived from MOA's agent class.

data(DestinationName n, Object o) - This function is overridden by the application-level concrete class derived from MOA's agent class. The application developer will provide the code that reads data from a communication channel. The parameter n is the source the data has arrive from and the parameter o is the data that is transmitted.

sendMessage(DestinationName nn, Object o) - Send a message to another local or remote agent or place. The parameter nn is the target of the message and the parameter is the message object that contains the content of the message.

receiveMessage(DestinationName nn, Object o) - This function is overidden by the application-level concrete agent class. This is the message handler for the messages for which the agent object is the target. The parameter nn is the name of the sender and the parameter o is the content of the message. The application code will handle the message based on the pre-defined type of the message.

3.2.2 Place State and Interface

Place State

As already mentioned, the place abstraction serves a couple of purposes. The first one is to provide a security boundary: All agents running within a place that has references to other places/agents/resources can share data with such objects. Mutual authorization verification between place and agent is performed before an agent is allowed to enter a place. The second purpose is to provide for additional granularity of resource management. Connections can be attached to a place, serving across agent migration (a place does not have to be deleted after an agent moves away from the place), allowing agents to leave repositories of data for subsequent agents that visit the same place.

Each agent must always reside in a place. An agent can have multiple places, which can remain after the agent moved from the node. This will happen when an agent anticipates that it will eventually return to the same node, or if it wants to leave some data for the agents that will visit this node at a later point of time. The pipelining type of applications has the need for remaining places. If the data is big enough, or it is only locally available, this might be the most appropriate way of data exchange. Similarly, there might be too many other agents interested in this information, or not all of them might be interested in it.The remained place will live only for the lifetime of the agent, after which it will also die. The reason might be that the agent will leave information for other agents.

A place in MOA does not expose its private internal state to the agent developer. However, the developer may add state variables to an application level abstraction of a place that is derived from MOA's place.

Place State

A place in MOA does not expose its state to the agent developer.

Place Interface

```
PlaceName getName()
AgentChannelAccess openChannel(DestinationName destName)
void receiveIncomingChannel(AgentChannelRequest)
void data(DestinationName, Object)
void closeOpenChannels()
Object sendMessage(DestinationName, Object)
Object receiveMessage(DestinationName, Object)
void log(String, String)
void takeDownAC(Failure)
Service getService(String)
```

Place Interface

getName()- Returns the name of the place.

openChannel(DestinationName nn) - Opens a communication channel to an agent pr a place. The parameter nn specifies the name of the other end of the channel. The channel is a java socket (streams-base) communication channel. Communication channels are discussed in detail below in Section 3.2.3.

receiveIncomingChannel(AgentChannelRequest) - This function is overidden by the application developer with application code that handles an incoming communication channel. This function is overidden by the application-level concrete class derived from MOA's place class.

data(DestinationName n, Object o) - This function is overidden by the application-level concrete class derived from MOA's place class. The application developer will provide the code that reads data from a communication channel. The parameter n is the source the data has arrive from and the parameter o is the data that is transmitted.

closeOpenChannels() - The current channels that are in an open state are closed.

sendMessage(DestinationName nn, Object o) - Send a message to another local or remote agent or place. The parameter nn is the target of the message and the parameter is the message object that contains the content of the message.

receiveMessage(DestinationName nn, Object o) - This function is overidden by the application-level concrete class. This is the message handler for the messages for which the place object is the target. The parameter nn is the name of the sender and the parameter o is the content of the message. The application code will handle the message based on the pre-defined type of the message.

log(String type, String dat) - A log of type typ is generated and logged with the logging service. The log information is encoded in the parameter dat.

takeDownAC(Failure) - Terminates an agent and destroys the place associated with the agent.

getService(String) - Returns back a service that is directly accessible to an agent or a place.

3.2.3 Communication Channels Interfaces

The communication channels are the mechanism that MOA provides agents and place to communicate with each other. The communicant channels in MOA are persistent over migration of an agent. A communication channel can be remote or local. Local channels use pipes and remote use sockets.

In general there are two ends to a communication channel the source and target. The source makes the request to open a channel to the target. The target may opt to accept the request or reject the request for a channel. The key abstractions in channel management are AgentChannelRequest and AgentChannelAccess. An AgentChannelRequest may be of type AgentIncomingChannel or OpenChannelRequest. An AgentIncomingChannel provides and interface by which a place may accept or reject a channel request. An openChannelRequest is analogous but in this case the target is an agent.

The source, an agent or a place, makes a request to open a channel via the openChannel(DestinationName nn) api call and receives an AgentChannelAccess object on return. The request is routed to the target in the form of an AgentChannelRequest and the target on receiving the request may choose to accept or reject the request. If the target accepts the request, then the target gets an AgentChannelAccess object. Note that both source and the target get back an AgentChannelAccess. An AgentChannelAccess is a handle to a communication channel via which the source and target may read/write data from/to each other.

The three abstractions AgentChannelRequest and AgentChannelAccess are discussed below.

<u>AgentChannelRequest State</u>
An AgentChannelRequest has no state and is a provider of services only.

<u>AgentChannelRequest Interface</u>

```
AgentChannelAccess accept()
void reject(Failure)
DestinationName getOriginatorName()
```

AgentChannelAccess accept() - The target of a request to open a channel accepts the request by a source to open a channel and gets a ChannelAccess object with which the target may read/write data to the communication channel.

void reject(Failure) - The target of a request to open a channel rejects the request to open a channel.

DestinationName getOriginatorName() - Returns the name of the source requesting the communication channel.

AgentChannelAccess State
There is no state of an AgentChannelAccess object that is exposed to the application.

AgentChannelAccess Interface

```
DestinationName getDestinationName();
void close();
void flush();
void write(Object data);
```

getDestinationName() - Returns the name of the source requesting for an open channel.

close() - Closes the channel associated with the AgentChannelAccess object.

flush() - The data in the communication channel is flushed.

write(Object data) - The data is written to an output stream which may then be read by the reader at the other end of the communication channel.

3.2.4 Agent Monitor State and Interface

Agent Monitor Interface

```
agentPlace locate(agent);
void kill(agent);
void killFamily(agent);
void killGeneration(agent);
void suspend(agent);
void resume(agent);
byte[] trace(agent, LOG_TARGET);
```

Agent Monitor State

TBD

Agent Monitor Interface

agentPlace **= *locate***(*agent)* - locate the place where this agent is currently executing. The name of the place also indicates the location of the agent environment. Note that this method, as well as other methods of an agent monitor, is executed at the location where an agent was created. Therefore, authentication of the owner is performed at the same place. Then the desired functionality (*locate*, *suspend*, etc.) is forwarded to the prospective nodes that contain the agent. It is responsibility of the locating schemes to locate agent and forward the desired action towards its current location (e.g. suspend, terminate, etc.). In the future, it would be possible to delegate ownership of the agent to some other owner. This would be achieved by passing the proxy structure, currently representing the agent on the originating node, to some other node. This is not envisioned at the moment, but it is considered in the future.

kill*(agent)* - kill the specified agent. Only the owner can invoke this method.

killFamily*(agent)* - Specifying any agent within a family suffices to determine and kill the whole family. Only the owner can invoke this method.

killDescendants*(agent)* - The whole generation subtree (below the given agent) is killed, e.g. the tree branch *agent.1.2.1*.x (including agent 1.2.1) will be killed. If the original agent is specified, then the whole generation is killed. Only the owner can invoke this method.

suspend*(agent)* - temporarily suspend the execution of this agent. It is resumed by a corresponding *resume* method. There also exist methods targeted at family and generation. Only the owner can invoke this method.

resume*(agent)* - resume an agent previously suspended by a *suspend* method. There also exist methods targeted at family and generation. Only the owner can invoke this method.

trace*(agent, LOG_TARGET)* - trace an agent execution. The same set of possible targets can be requested, as in case of logging. The difference is that tracing is performed interactively, by requiring agent to forward back to the home node all logged data as soon as they are encountered. The logged data is stored in the agent proxy at the home node. There also exist methods targeted at family and generation. Only the owner can invoke this method.

3.2.5 Nameserver State and Interface

The nameserver plays the role of tracking the location of an agent as it migrates across one or more nodes. The nameserver maintains the local information about an agent and provides it to the locator upon request. The nameserver supports an interface for registering and unregistered an agent location as well as support agent location lookup. An agent lookup can provide full information about an agent location in the form of a location object, or it may provide only the host:port pair of the current agent location. Location information about an agent is maintained only during the lifetime of an agent and discarded thereafter.

The nameserver supports multiple strategies for registration of an agent location. The default strategy always registers the agent location at home after each successful move. A second strategy is to register the agent location at a pre-defined server after each move. A third strategy requires registering the agent location at the source node prior to moving to a target node. The fourth strategy is to search for the agent itinerary for a possible location of the agent. The nameserver interface permits the client to specify a strategy during registration.

Nameserver State
The nameserver does not have state that is exposed to the agent developer.

Nameserver Interface

```
void setCacheSize(int);
Location register(String, URL, int, URL[], long);
void unregister(String);
URL lookupAgent(String);
Location lookupAgentLocation(String);
Enumeration lookupAgentFamily(String);
Enumeration listAllAgents();
```

setCacheSize(int nn) - Sets the size of the cache in which the location objects are stored.

register(String nn, URL loc, int st, URL[] iter, long tt) - Register the agent location. The parameter nn is the name of the agent, parameter loc is the agent location, parameter st specifies the strategy for location, iter specifies the itinerary of the agent, and tt is the agent's time to live. The function returns a Location object, if there exists a previously registered Location object.

unregister(String str) - Unregister the agent named str at the local nameserver.

lookupAgent(String nn) - Returns the agent's current location if it is registered at the nameserver. The agent name is specified by the parameter nn, and the return value is the URL of the nameserver where the agent was last located.

listAllAgents(String nn) - Looks up all the agents registered at this nameserver and returns an Enumeration of agent names.

lookupAgentLocation(String nn) - Looks up the location of an agent registered with the nameserver and returns the location object. The agent names is specified by the parameter nn.

lookupAgentFamily(String nn) - Lookup the family of the agent specified by the name nn and return an Enumeration of all agents that belong to that family that are registered at the nameserver. The enumeration contains agent names.

3.2.6 Hierarchical Nameserver State and Interface

Hierarchical Nameserver State

The hierarchical nameserver does not expose state to the agent developer

Hierarchical Nameserver Interface

register(ServerName xx, ServerName yy) - Registers the nameserver yy as the child of nameserver xx. The function returns the tree hierarchy to the top.

unregister(ServerName xx, ServerName yy) - Unregisters the nameserver yy that is registered with the nameserver xx.

lookupTree(ServerName xx, AgentName yy) Returns the location of the agent yy if it is located anywhere in the sub-tree of the nameserver xx. The return value is a location object.

returnChildren(ServerName xx) - Returns the children nameservers that are registered with the nameserver xx. The return value is an enumeration of the child nameservers.

returnTree(ServerName xx) -Returns the whole subtree under the nameserver xx. The return value in an enumeration of the child nameservers.

3.2.7 User State and Interface

The user object serves as an interface to a user applet, tracking information of interest to a user, allowing an operator to run the applet repeatedly without the need to re-enter the operator's specific interest each time.

User State

The state is not accessible to the application developer for read or write operations.

User Interface

The bulk of the work done by user is in sending and receiving messages. The messages are (1) logon; (2) Launch an agent; (3) Get Family; (4) Update Family; (5) Queries on logs and snapshots, etc. The user object provides the message handlers to service the messages.

3.2.8 Home Service

Home service is a repository for snapshots and logs of all agents which that site serves as a home or alternate home. The home service holds the agent's policy so that it is able to track the data which an agent is permitted to access.

Home Service State

The state of the home service object is not accessible to the agent developer.

Home Service Interface

The home service delivers its services by sending and receiving message to and from other moa components such as the logger. The message that are handled by homeservice range from accepting an agent property object for an agent to routing an agent log to the logger. The home service also handles queries on the repository of snapshots that it maintains.

3.2.9 Batch Driver

The Batch driver is a script that provides a command line interface to the MOA system. The script enables the operator to logon to MOA, launch an agent, display the agent family, display all the launched agents, update the agent family, list and fetch snapshots, display the home of the agent, etc.

3.2.10 Open Issues

Should we, and if so how, support relationships between agents. For example, it would be useful to control a number of agents and then kill/suspend/control them all at once. I guess that this is a useful feature. At the moment, this is supported for agents of the same origin (the same family). It would be useful to allow this kind of support for dynamic relationship of arbitrary agents of arbitrary origin, even class.

Nested places. As already discussed in Section 3.2.2. the nesting of places can provide some additional encapsulation of security and resources at the place level. The main disadvantage is in extended complexity. One more time we shall re-evaluate the tradeoffs before making final decision on introducing nested places.

3.3 System properties

The final target platform is x86 PCs connected to the Internet.The project will also attempt to use a couple of PCs located in Europe and Japan. When fully developed, the system should be capable of encompassing the whole Web (modulo security restrictions).

Performance requirements are not strict, they will largely depend on the scale of the target area that agent will visit and the number of agents. The actual scalability numbers will be determined once that the first prototypes are developed. The following metrics will be relevant: How many agents/places can be supported on a node; how many agents can be transferred during a given time between two nodes; The speed and efficiency in locating agents that migrated; the amount of state required to maintain the agent state and its resources.

3.4 Constraints

3.4.1 Real Time

There will be no attempt to make any support that would satisfy real-time constraints. Nevertheless, by its own nature, agent could improve real-time behavior of some applications by virtue of moving towards the source of the communication.

3.4.2 Security

Due to the lack of security support in the initial version, initial deployment of agents will not be possible through firewalls, unless there exists some additional support provided by such environments. Nevertheless, some sort of security mechanisms will be deployed, subject to their availability in the Java and other tools.

3.4.3 Fault management

There will be no attempt to make any support that would satisfy fault management. Nevertheless, by its own nature, agent could improve fault management behavior of some applications by virtue of being able to move away from the node that has partially failed.

3.4.4 Configurability

The agent system will be a relatively small software package. Nevertheless, a number of system parameters will be possible to change dynamically. For example, the default agent life-time, the maximum number of various objects, such as agents, places allowed at an agent environment at one time, etc. Nevertheless, not all of these parameters will be dynamically configurable.

3.4.5 Administration

It will be possible to administer the local agent environment and the agent monitor, but not the collection of remote nodes (although it would be possible to administer remote nodes one at a time).

3.4.6 Hardware

There should be no constraints imposed on the hardware. The advantage of using Java is portability, so our agent environment will be limited only by the portability of the Java VM to different hardware platforms.

3.5 Development requirements

3.5.1 Build environment

Building of software will be perform periodically on the all files that group will develop. A common build tree will be maintained for all participants, probably in the NFS space that all project participants will have access to.

3.5.2 Tools

The project will use one of the two development tools, either Java Workshop, or Symantec Cafe.

3.5.3 Hardware

Development will be pursued on PC platforms because of the availability, prices and quality of the tools for Java development. Periodically, the developed software will be run on the target platform, HP machines.

3.5.4 Software

The available Java compilers/interpreters will used. It will either be Symantec JIT or JDK toolkit.

4 Deferred Features

4.1 Deferred features

Security is one of the most important deferred features. There will be no specific work in the security in this project.

4.2 Explicit restrictions

Initial deployment of agents will not be possible through firewalls, unless there exists some additional support provided by such environments. See also Section 3.4.2.

5 QA

5.1 Criteria/Metrics

Considering that this is not a product version, but advanced prototype there is no sophisticated QA criteria/metrics planned at this time. The modules will be tested for functionality and for performance requirements to be determined during development, subject to the compiler type and the scale of testbed and application.

5.2 Unit Tests

We shall test for functionality and performance of each abstraction, method and module. In some cases testing will be planned already during development. i.e. test will be provided and integrated during development. Some of the test will be provided later.

5.3 Reference to QA test plan

There is no immediate QA test plan. The testing and quality assurance is conducted during development and after implementation.

6 Documentation

6.1 New manuals required

In addition to architectural, functional and design specification, there will be programmers and users manual. Programmers manual will specify in more detail functional API and users manual will specify user interface. Programmer's manual will

6.2 Manuals requiring modification

It will be possible that architectural specification will undergo some changes in the way how some of the modules are organized. The same applies to functional specification which may adjust to the

Appendix A Initial List of Risks/Issues

We have intentionally avoided any dependency on emerging technologies, such as remote method invocation. On the other hand, we did closely follow the new functionality provided by JDK 1.1.

Appendix B Open Issues

A number of open issues were referenced throughout the text.

Appendix C Batchdriver Commands

The batchDriver was developed that support the batch interaction with the MOA system. Detailed description of batchDriver is beyond the scope of this document, only the supported commands are described in the following text.

listSnapshots *<host> <port> <launchName>* - lists the agentName and nos for the entire Vector of snapshots for the given launchName;.

fetchSnapshot *<host> <port> <agentName> <snapshot sequenceNo>* - will get a particular snapshot from the user's HomeService and set the class attribute "currentSnapshot". It extracts the family object from the JarFile and sets the class attribute "currentFamily"

createPCA *<fileName>* creates a PCA file at the user's node under the input fileName using the currentSnapshot set as a result of the command "fetchSnapshot"

showFamily *<filename>* gets the family object for the specified fileName and displays its contents. If the <fileName> is not specified it shows the contents of the currentFamily

trace *<on/off>* - switches on/off the trace of the currentFamily object

countFrequency *<frequency>* sets the countFrequency of the currentFamily

counts - configures the counts for the currentFamily

watch - configures the watches for the currentFamily

updateFamily - uses the currentFile and currentFamily objects and sends a request to the user to update the currentFile with the currentFamily object. Note: currentFamily and currentFile can be set using the fetchSnapshot or showFamily commands

showHome *<agentName>* shows the home and alternate home URLs of the specified agent. Instead of showing the url's, lets show the AEname (host & port). When no agent name is specified, lets take the info from the current snapshot.

queryLogger - queries the logger at a specified host and port.

showpca - shows the list of pca (preconfigured agents) files in the user's directory.

showla - shows the list of la (launched agents) files in the user's directory.

logon - is for logging on of operator.

launch - enables the launching of preconfigured agents.

end -exits

pause *<delay>* makes the operator thread to sleep for the specified delay.

NOTE: The following commands showFamily <filename> (if filename is not specified it uses the currrentFamily), trace, countFrequency, counts, watch, updateFamily, fetchSnapshot, and createPCA operate on two class attributes (currentFamily, currentFile - which is the current JarFile). These two attributes can be set in two ways:

showFamily *<filename>* Here we use the filename to set the currentFile, and get the corresponding Family object to set currentFamily.

fetchSnapshot *<agentName> <snapshot sequenceNo>* Here we use these arguments to fetch a particular snapshot. Using the snapshot operator can use the "createPCA >filename" command to save the snapshot as a PCA file (which is the currentFile). The currentFamily attribute is set by extracting the family object from the snapshot JarFile

logTypes *<type>* Adds the LogType corresponding to input string. The Family objects maintains a hashtable of LogType objects where the String representation of LogType is the key.

executeCmd *<cmdType> <agentName>* Executes a command on the specified agent. CmdType can be *<SuspendAgentMvmt>* which suspends agent movement, *<Suspend-*

AgentThread>, which suspends the execution of the agent thread, and *<ResumeAgent>* resumes previously suspended agent.

listLocalLogs lists the logs in the current snapshot.

Acknowledgments

The following people read the document and provided a feedback that significantly improved its presentation and contents: David Black and Shai Guday.

Bibliography

[Genm] General Magic White Paper "Telescript Technology: Mobile Agents", available as *http://www.genmagic.com/Telescript/Whitepapers/wp4/whitepaper-4.html.*

[URI94] Universal Resource Identifiers in WWW, Network Working Group, RFC 1630, June 1994.

[URL94] Uniform Resource Locators, Network Working Group, RFC 1738, December 1994.

[URN94] Functional Requirements for Uniform Resource Names, Network Working Group, RFC 1737, December 1994.

[Whit] Jim White, General Magic Inc., "Prospectus for an Open Simple Agent Transfer Protocol", *available as http://www.agent.org/pub/satp/prospectus3.html.*