Joint Submission

Mobile Agent System Interoperability Facilities Specification

GMD FOKUS
International Business Machines Corporation

Supported by:
Crystaliz, Inc.
General Magic, Inc.
The Open Group

November 10, 1997
OMG TC Document orbos/97-10-05

The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013.

OMG and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB CORBA, CORBAfacilities, and CORBAservices are trademarks of the Object Management Group.

Java is a trademark of Sun Microsystems, Inc.

Aglets is a trademark of International Business Machines Corporation.

Other names, products, and services may be the trademarks or registered trademarks of their respective holders.

Table of Contents

Common Conceptual Model 1

Mobile agents (also called transportable agents) are a relatively new technology that is fueling a new industry. Because the technology and the industry are new, mobile agent systems (for example, Crystaliz's MuBot, Dartmouth College's AgentTcl, IBM's Aglets, the Open Group's MOA, GMD FOKUS's JMAF/Magna, and General Magic's Odyssey) differ widely in architecture and implementation.

The differences among mobile agent systems prevent interoperability and rapid proliferation of agent technology, and has probably impeded the growth of the industry. To promote both interoperability and system diversity, some aspects of mobile agent technology must be standardized.

This chapter provides an introduction on interoperability and describes the terminology of this submission. In addition, the basis of current commonality in mobile agent systems is provided in the areas of agent interaction, agent transfer, and security. Following the definition of a common conceptual model, sets of features are extracted which will be specified to promote interoperability and enhance the functionality of most existing mobile agent systems. Together, the definitions and features provide a common interoperability base for mobile agent technology.

1.1 Interoperability

An important goal in mobile agent technology is interoperability between various manufacturer's agent systems. Interoperability becomes more achievable if actions such as agent transfer, class transfer, and agent management are standardized. When the source and destination agent systems are similar, standardization of these actions can result in interoperability. However, when the two agent systems are dramatically different, only minimal interoperability can be achieved.

Interoperability in this document is not about language interoperability. Mobile Agent System Interoperability Facilities (also called MAF, an acronym for the original proposal, Mobile Agent Facility) is about interoperability between agent systems written in the same language, but potentially by different vendors and systems that are expected to go through many revisions within the life time of an agent. Language interoperability for active objects that carry “continuations” around is technically difficult to achieve. Furthermore, it is not needed, since the support for different languages can be replicated at each node.

This specification does not define standardization of local agent operations such as agent interpretation, serialization, execution, or deserialization. However, these actions are implementation specific, and there is currently no compelling reason to limit agent system implementations to a single architecture.

1.1.1 What Should Be Standardized Now?

There are several areas of mobile agent technology that the mobile agent community should standardize now to promote interoperability:

- Agent management
- Agent transfer
- Agent and agent system names
- Agent system types
- Location syntax

This section discusses the reasons for standardizing these aspects of mobile agent technology now.

Agent Management

There is interest within the mobile agent community to standardize agent management. One can envision a system administrator managing agent systems of different types via standard operations. It should be possible to create an agent given a class name for the agent, suspend an agent’s thread of execution, resume its thread, or terminate it in a standard way. Defining common management functions permits one administrator to manage agent systems of various types.

Agent Transfer

It is advantageous for two agents to communicate at the same location rather than across a network for two reasons: number of network transactions, and data monitoring. Allowing a source agent to travel to an agent system close to the destination agent system achieves the benefit of locality.

If the network between the two agents has low bandwidth, filtering data across the net can be expensive and time consuming when compared to local data transfers. For example, if the two agents are using RPC, multiple operations across the network are usually necessary to transfer the data that one agent needs from the other.

Data monitoring (for example, waiting for a certain stock to go up ten points) is a task that can continue for days. It may be preferable to send a mobile agent to the platform providing the data, rather than use a stationary agent to send periodic inquiries for the latest stock price. For this type of application, using a mobile agent is cost- and resource-efficient.

Agent and Agent System Names

In addition to standardizing operations for interoperability between agent systems, the syntax and semantics of various parameters must be standardized. Specifically, agent name, agent system name, and location should be standardized.

When invoking a management operation, the agent being managed must be identified. Therefore, the agent name syntax should be standardized. A standardized agent name syntax also provides two other benefits. It allows an agent system to quickly determine whether it can support an incoming agent, and it allows two agents to identify each other by name.

Agent System Type and Location Syntax

The location syntax must be standardized so that an agent can access agent system type information from the desired destination agent system, and so that the source and destination agent systems can identify each other. If the agent system type can support the source agent, the agent transfer can happen.

It is also important to provide a naming authority (an organization that assigns a unique identifier) for each agent system type. Ensuring uniqueness of agent system type names prevents two companies from duplicating agent system type values.

1.1.2 What Should Be Standardized Later?

The previous section discussed the aspects of mobile agent technology that should be standardized now to ensure agent system interoperability. There are other aspects of the technology that should be standardized, but not until the industry is more mature, or related de facto standards are available. This section describes some of the areas that will need standardization in the future.

When an agent takes a multi-hop travel, which travels between more than two security domains (regions, see Section 1.2.13, “Regions“), the security issues become complex. Most security systems today deal only with security between two domains, which is single-hop travel. The mobile agent community should delay standardizing multi-hop security of mobile agents until security systems can handle the problem.

Today’s mobile agent systems use several different languages (for example, Tcl and Java). Therefore, the effort to convert from one agent encoding to another is too complex. When the serialization formats for agent code and execution state are similar, it should be possible to build standard bridges between different agent system types. In the future, features are likely to be added to MAF that improve interoperability, and minimize the need for bridges.

1.1.3 MAF Interoperability Summary

Interoperability is an important goal of the MAF Specification. Table 1-1 describes the types of interoperability MAF addresses, and estimates the complexity that agent systems require to support it.

Agent management allows an agent system to control agents of another agent system. MAF addresses this interoperability by defining interfaces for actions such as suspending, resuming, and terminating agents. This interoperability is relatively straightforward for most agent systems to implement.

Agent tracking permits the tracing of agents registered with MAFFinders (i.e. Naming Services) of different agent systems. This interoperability is also relatively straightforward for most agent systems to implement.

Agent communication is outside the scope of the MAF specification. It is extensively addressed by CORBA as object communication. Therefore, agent communication is omitted from this MAF specification.

MAF addresses agent transport by defining methods for receiving agents and fetching their classes. Agent transport interoperability is not simple to achieve and requires a certain amount of cooperation between implementors of different agent systems to achieve.

Table 1-1

Function	Addressed by MAF	Level of Complexity to Support
agent management	YES	straightforward
agent tracking	YES	straightforward
agent communication	NO	N/A
agent transport	YES	complex

Note that this chapter presents an overview of interoperability and does not discuss the details of achieving interoperability. These details are discussed in Chapter 2, “CORBA Services”, and Chapter 3, “MAF IDL”.

An important goal in mobile agent technology is interoperability between various manufacturer’s agent systems. Interoperability becomes more achievable if actions such as agent transfer, class transfer, and agent management are standardized. When the source and destination agent systems are similar, standardization of these actions can result in interoperability. However, when the two agent systems are dramatically different, only minimal interoperability can be achieved.

1.2 Basic Concepts

This section defines the major mobile agent concepts. Note that this specification uses object terminology to describe mobile agent concepts. For mobile agent systems that are not object oriented, substitute the word ‘code’ for class in the definitions.

1.2.1 Agent

An agent is a computer program that acts autonomously on behalf of a person or organization. Currently, most agents are programmed in an interpreted language (for example, Tcl and Java) for portability. Each agent has its own thread of execution so tasks can be performed on its own initiative.

1.2.2 Stationary Agent

A stationary agent executes only on the system where it begins execution. If the agent needs information that is not on that system, or needs to interact with an agent on a different system, the agent typically uses a communications transport mechanism such as Remote Procedure Calling (RPC).

The communication needs of stationary agents are met by current distributed object systems such as CORBA, DCOM, and RMI.

1.2.3 Mobile Agent

A mobile agent is not bound to the system where it begins execution. It has the unique ability to transport itself from one system in a network to another. This submission is primarily concerned with mobile agents.

The ability to travel permits a mobile agent to move to a destination agent system that contains an object with which the agent wants to interact. Moreover, the agent may utilize the object services of the destination agent system.

Although current distributed object systems meet the communications needs of stationary agents, they do not meet the communications needs of mobile agents. A mobile agent has more capabilities and requirements than many current distributed object systems can address at this time.

1.2.4 Agent State

When an agent travels, its state and code are transported with it. In this context, the agent state can be either it’s execution state, or the agent attribute values that determine what to do when execution is resumed at the destination agent system. The agent attribute values include the agent system state associated with the agent (e.g. time to live).

1.2.5 Agent Execution State

An agent’s execution state is its runtime state, including program counter and frame stacks.

1.2.6 Agent Authority

An agent’s authority identifies the person or organization for whom the agent acts. An authority must be authenticated.

1.2.7 Agent Names

Agents require names that can be identified in management operations, and can be located via a naming service. Agents are named by their authority, identity and agent system type. An agent’s identity is a unique value within the scope of the authority that identifies a particular agent instance. The combination of an agent’s authority, identity and agent system type is always a globally unique value. Because an agent’s name is globally unique and immutable, the name can be used as a key in operations that refer to a particular agent instance.

1.2.8 Agent Location

The location of an agent is the address of a place (refer to Section 1.2.12, “Place“ for more information). A place resides within an agent system. Therefore, an agent location should contain the name of the agent system where the agent resides and a place name. Note that if the location does not contain a place name, the destination agent system chooses a default place.

1.2.9 Agent System

An agent system is a platform that can create, interpret, execute, transfer and terminate agents. Like an agent, an agent system is associated with an authority that identifies the person or organization for whom the agent system acts. For example, an agent system with authority Bob implements Bob's security policies in protecting Bob's resources.

An agent system is uniquely identified by its name and address. A host can contain one or more agent systems. Figure 1-1 on page 7 describes an Agent System.

1.2.10 Agent System Type

An agent system type describes the profile of an agent. For example, if the agent system type is Aglet, the agent system is implemented by IBM, supports Java as the Agent Language, uses Itinerary for travel, and uses Java Object Serialization for its serialization. This specification recognizes agent system types that support multiple languages, and languages that supports multiple serialization methods. Therefore, a client requesting an agent system function must specify the agent profile (agent system type, language, and serialization method) to uniquely identify the desired functionality.

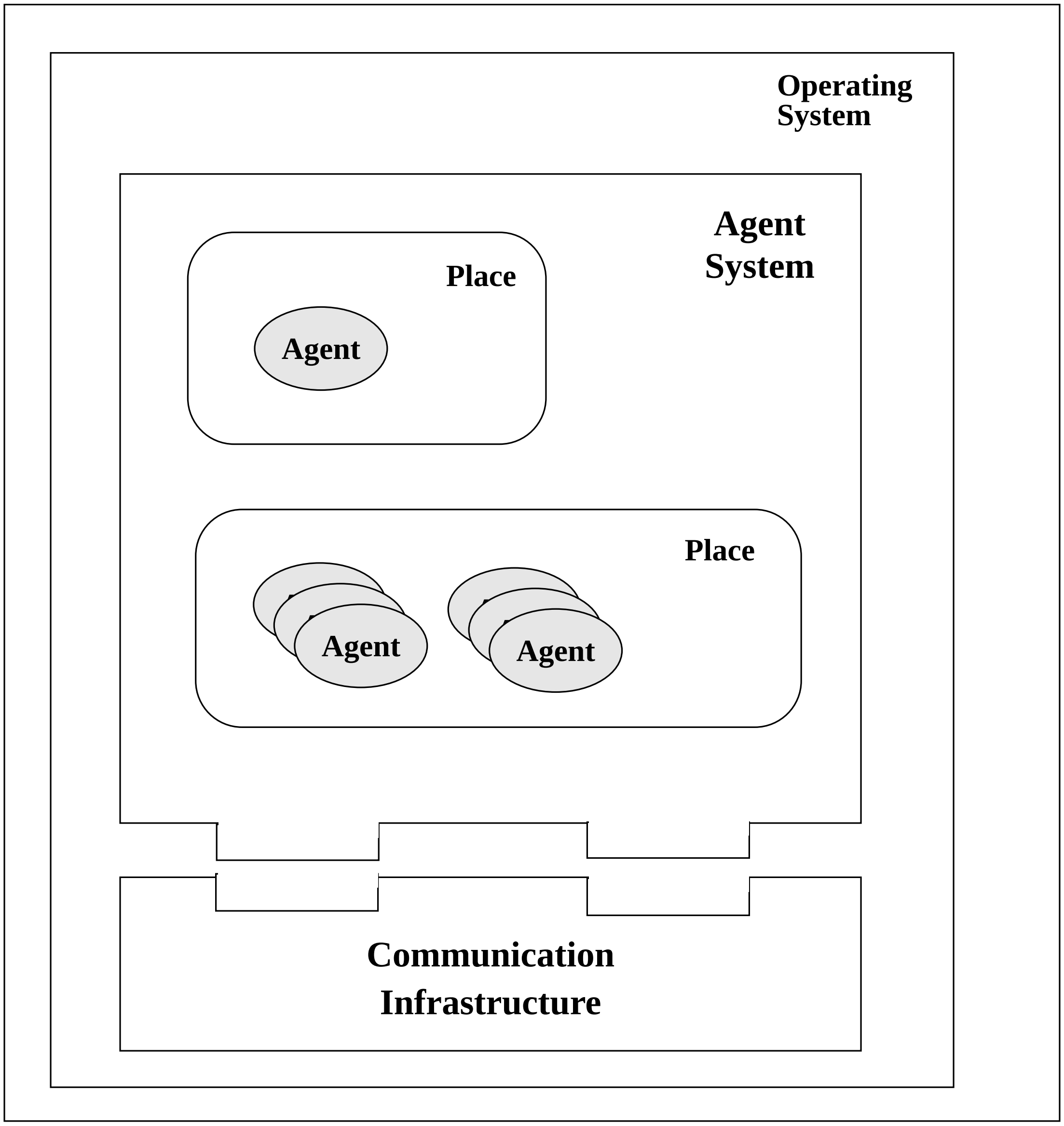


Figure 1-1 Agent System

1.2.11 Agent System to Agent System Interconnection

All communication between agent systems is through the Communication Infrastructure (CI). The region administrator defines communication services for intra-region and inter-region communications. Figure 1-2 on page 8 describes Agent System to Agent System Interconnection.

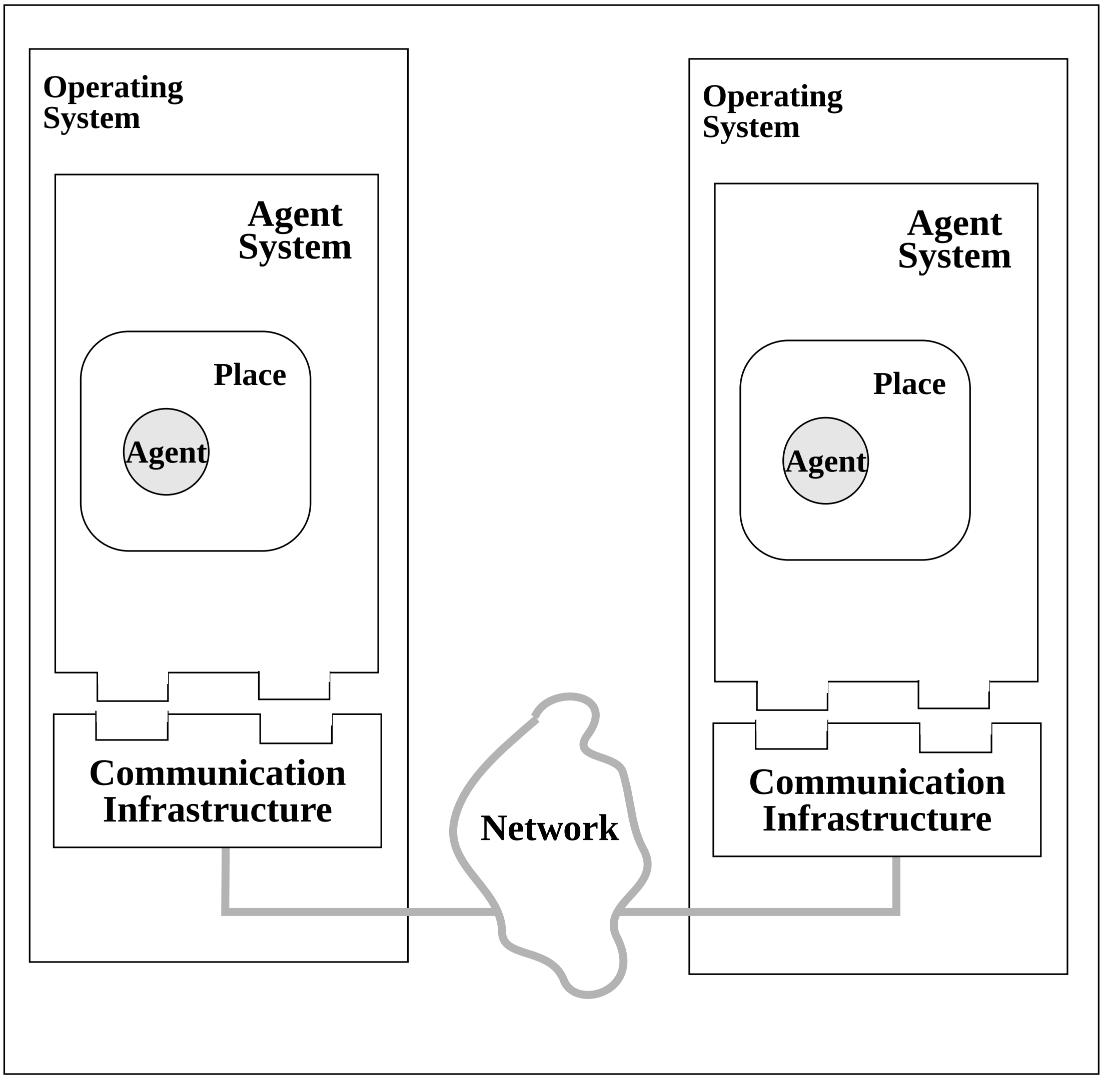


Figure 1-2 Agent System to Agent System Interconnection

1.2.12 Place

When an agent transfers itself, the agent travels between execution environments called places**.** A place is a context within an agent system in which an agent can execute. This context can provide functions such as access control. The source place and the destination place can reside on the same agent system, or on different agent systems that support the same agent profile.

A place is associated with a location, which consists of the place name and the address of the agent system within which the place resides. An agent system can contain one or more places and a place can contain one or more agents. Even though a place is defined as the environment where an agent executes, if agent system does not implement places, then place is still defined as the default place.

When a client requests the location of an agent, it receives the address of the place where the agent is executing.

1.2.13 Regions

A region is a set of agent systems that have the same authority, but are not necessarily of the same agent system type. The concept of region allows more than one agent system to represent the same person or organization. Regions allow scalability because you can distribute the load across multiple agent systems.

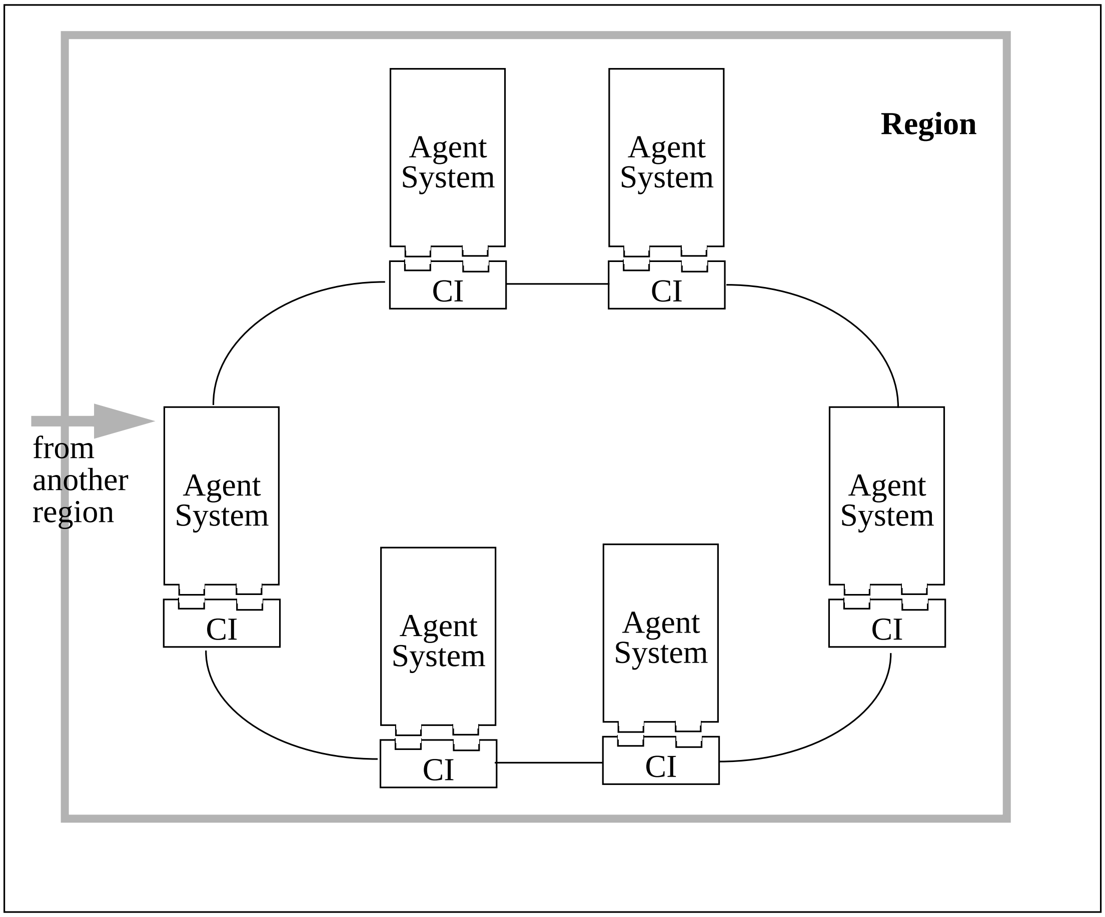


Figure 1-3 Region Architecture

A region provides a level of abstraction to clients communicating from other regions. A client wishing to contact an agent or agent system may not be aware of the agent's location. Instead, a client has an address for the region (basically, the address of an agent system that is designated as the region access point), and the name of the agent or place. It is now possible to contact and communicate with an agent or agent system with only this information.

An agent may also have the same authority as the region in which it is currently residing and executing. This means that the agent represents the same person or organization as the region. Normally, the configuration of the region may grant a richer set of privileges to such an agent than to another resident agent with a different authority. For example, an agent that has the same authority as the region may be granted administrative privileges. A region can be same as an identity domains of CORBA security if the authority of the region equals to identity of the identity domain.

A region fully interconnects agent systems within its boundaries and enables the point-to-point transfer of information between them. Each region contains one or more region access points and by these means, regions are interconnected to form a network. Figure 1-3 on page 10 describes region architecture.

1.2.14 Region to Region Interconnection

Regions are interconnected via one or more networks and may share a Naming Service based on an agreement between region authorities and the specific implementation of these regions. A non-agent system may also communicate with the agent systems within any region as long as the non-agent system has the authorization to do so.

A region contains one or more agent systems. Agent systems and clients outside of the region access the region via agent systems that are exposed to the outside world, similar to a firewall situation. These agent systems are defined as region access points.

Access rights are allocated to agents, based on the same authority as the region in which they are currently running. Using this definition, a region is regarded as a security domain in the context of MAF. Figure 1-4 on page 12 describes region to region interconnection.

1.2.15 Serialization/Deserialization

Serialization is the process of storing the agent in a serialized form. Deserialization is the process of restoring the agent from its serialized form.

The key to storing and retrieving agents is representing the state of an agent in a serialized form that is sufficient to reconstruct the agent. Note that the serialized form must be able to identify and verify the classes from which the fields were saved.

For agent systems that are not object oriented, the agent state is the extraction of runtime data for the agent, and classes are the code that implements the agent.

1.2.16 Codebase

Codebase specifies the location of the classes used by an agent. The codebase can be agent systems or non CORBA object such as Web servers. If an agent system is responsible for providing the necessary classes, the codebase must have enough information to locate the agent system. Such agent system is called **class provider**.

It is possible for an agent to move to such an agent system where the codebase is not directly accessible (For example, due to the firewall.) Refer to “Class Transfer“ in Section 1.3.1, “Remote Agent Creation“ for more information.

1.2.17 Communications Infrastructure

A communications infrastructure provides communications transport services (e.g. RPC), naming, and security services for an agent system.

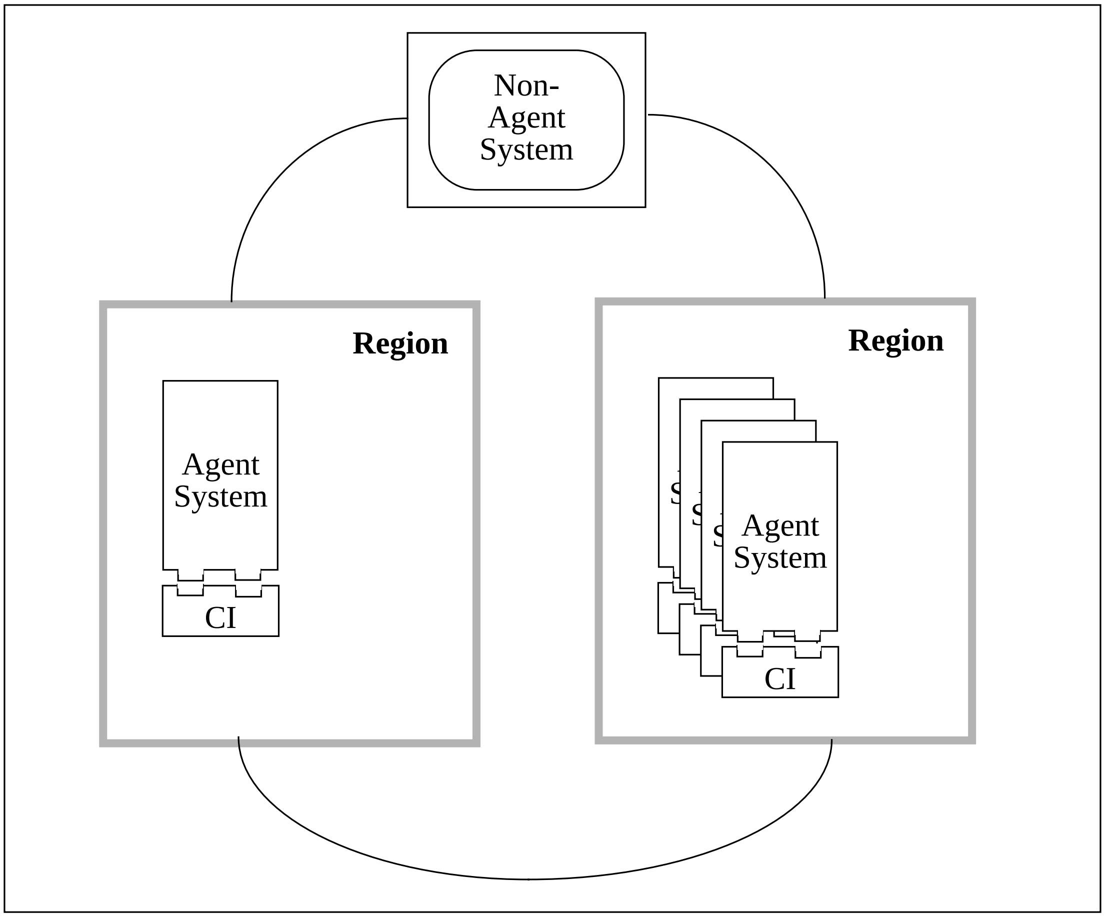


Figure 1-4 Region to Region Interconnection

1.2.18 Locality

In the context of mobile agents, locality is defined as being close to the destination agent system either in the same host or the same network.

1.3 Agent Interaction

Three common types of agent interactions are defined that are related to interoperability:

- remote agent creation
- agent transfer

- agent method invocation.

Note that only agent transfer is mobile-agent specific.

1.3.1 Remote Agent Creation

In remote agent creation, a client program interacts with the destination agent system to request that an agent of a particular class be created. A client program is:

- a program in a non-agent system
- an agent in an agent system of a different type than the destination agent system
- an agent in an agent system of the same type as the destination agent system

The client authenticates itself to the destination agent system, establishing the authority and credentials that the new agent will possess. The client supplies initialization arguments and, if necessary, the class needed to instantiate and execute the agent.

An agent system can also choose to create agents on its own initiative. The new agent will generally have the same authority as the agent system.

1.3.2 Agent Transfer

When an agent transfers to another agent system, the agent system creates a travel request. As part of the travel request, the agent provides naming and addressing information that identifies the destination place. The agent also specifies a quality of communication service required for agent transfer. The quality of communication service is not specified by MAF standard, it is left open to agent system implementors to specify it.

If the source agent system reaches the destination agent system, the destination agent system must fulfill the travel request, or return a failure indication to the agent. If the source agent system cannot reach the destination agent system, then a failure indication must be returned to the source agent system.

When the destination agent system agrees to the transfer, the source agent's state, authority, security credentials, and, if necessary, its code are transferred to the destination agent system (refer to "Class Transfer" in Section 1.3.1, "Remote Agent Creation" for a discussion of when it is necessary to transfer code). The destination agent system reactivates the source agent, and then execution is resumed.

1.3.3 Agent Method Invocation

An agent invokes a method of another agent or object, if authorized and has a reference to the object. As with agent transfer, the agent specifies the required level for quality of service. Similarly to agent transfer, this is not specified by MAF. Instead it is left open to application developers to specify it. The communications infrastructure must invoke the indicated method and return the result of the invocation, or return a failure indication. When an agent invokes a method, the security information supplied to the communications infrastructure executing the method invocation must be the agent's authority. Note that most distributed object systems currently support this function.

1.4 Functions of an Agent System

Common actions among agent systems are:

- Transferring an agent, which can include initiating an agent transfer, receiving an agent, and transferring classes
- Creating an agent
- Providing globally unique agent names and locations
- Supporting the concept of a region
- Finding a mobile agent
- Ensuring a secure environment for agent operations

The remainder of this section discusses these areas, defining terms as necessary.

1.4.1 Transferring an Agent

In the previous section, transferring an agent includes initiating the transfer, receiving an agent, and transferring classes. The following sections discuss each of these steps.

Initiating an Agent Transfer

When a mobile agent is preparing for a trip, the agent must be able to identify its destination. If the place is not specified, the agent executes in the default place of the destination agent system.

When the location of the destination agent system is established, the mobile agent requests the source agent system for a transfer to the destination agent system. This message is relayed using an internal API between the agent and the source agent system.

When the destination agent system receives the agent's trip request, the following actions are initiated:

- Suspend the agent (halt the agent's execution thread)
- Identify the pieces of the agent's state that will be transferred
- Serialize the instance of the Agent class and state
- Encode the serialized agent for the chosen transport protocol
- Authenticate client
- Transfer the agent

Receiving an Agent Transfer

Before an agent is received into a destination agent system, the destination agent system must determine whether it can interpret the agent. If the agent system can interpret the agent, it accepts the agent, then:

- Authenticates client
- Decodes the agent
- Deserializes the Agent class and state
- Instantiates the agent
- Restores the agent state
- Resumes agent execution

Class Transfer

Class transfer is the ability to transfer class information from one agent system to another. This ability is a requirement in agent systems that support object-oriented agents. Not all agents are object-oriented programs (for example, AgentTcl).

There are three reasons why class transfer is necessary during the life span of a mobile agent.

- Agent instantiation as a part of remote agent creation (Agent class needed)

  When an agent is created remotely by invoking a create operation at the agent system, the Agent class is needed to instantiate the agent. If the Agent class does not exist at the agent system that creates the agent, the class information must be transferred from the source agent system.

- Agent instantiation as a part of agent transfer (Agent class needed)

  After an agent travels to another agent system, the Agent class is needed to instantiate the agent. If the Agent class does not exist at the destination agent system, the class must be transferred from the source agent system.

- Agent execution after instantiation (classes other than Agent class needed)

  After an agent is instantiated due to remote creation or agent transfer, the agent often creates other objects. Obviously, the classes of these objects are needed for their instantiation. If any of these objects' classes are not available at the agent system that creates or receives the agent, they must be transferred from the source agent system.

The common conceptual model is flexible enough to support variations of class transfer so that implementors have more than one method available. Specifically, the model supports:

- **Automatic transfer of all possible classes**

  The source agent system (the class provider or the agent sender) sends all classes needed to execute the agent with each remote agent creation or transfer request. This approach eliminates the need for the destination agent system to request more classes. However, automatically sending all classes consumes more bandwidth than necessary if any of the transferred classes are already cached at the destination agent system.

- **Automatic transfer of the Agent class only, other classes transferred on demand**

  The source agent system sends the class needed to instantiate the agent with each remote agent creation or transfer request. If more classes are needed after instantiating the agent, the destination agent system issues requests for these classes to the class provider. If the class provider is not directly accessible from the destination agent system, the destination agent system issues the request to the sender agent system by calling *fetch_class* method along with the codebase of the classes. In this case, the sender agent system must be able to locate the requested classes either by using the codebase information, or by sending a further request to another agent system associated with the codebase. The sender agent may have a cache for the classes.

  This approach does not require the source agent system to determine all possible classes necessary before creating or transferring an agent. It is also more efficient as more classes are cached at the destination agent system. However, the agent creation or transfer request fails if the destination agent system cannot access the source agent system to transfer the necessary classes. This failure could happen, for example, if the source agent system is a portable computer that has been disconnected since the agent creation or transfer request was sent successfully.

- **Automatic transfer of the Agent class and on-demand transfer of all other classes when transferring an agent, coupled with transferring all classes automatically when creating an agent remotely**

  This approach is a combination of the first two. When a remote agent creation is launched by a client that is not always connected (for example, a laptop computer). Therefore, if all classes are automatically transferred for remote agent creation operations, then the problem of losing access to the classes available at the source agent system may be avoided.

- **Transfer a list of the names of all possible classes with the agent creation or transfer request**

  This approach is another combination of the first two. The source agent system sends a list of class names that includes all the classes necessary to perform the specific agent operation. The destination agent system then requests only the classes on that list that have not been cached. This approach is efficient, but still requires the source agent system to know which classes the agent needs before making the agent creation or transfer request. This technique is most effective when there is cooperation between the agent language compiler and the agent system. For example, the compiler can provide a list of classes associated with each agent so the agent system does not parse the agent's code at run time to get this information.

When an agent system requests a class transfer, the agent system must identify the class to another agent system.

1.4.2 *Creating an Agent*

For each agent, there is a class from which the agent system instantiates an agent. This class is defined as Agent. To create an agent, an agent system creates an instance of the Agent class within a default place or a place the client application specifies. The Agent class specifies both the interface and the implementation of the agent. To create an agent, an agent system should:

- Start a thread for the agent
- Instantiate the Agent class
- Generate (if necessary) and assign a globally unique agent name that can be authenticated
- Start execution of the agent within its thread

Since an agent executes on its own thread, the agent executes independently from all other agents.

1.4.3 Providing Globally Unique Names and Locations

An agent system must generate a unique name for itself, and the places it creates. It also have to generate a unique name for the agents it creates if required.

1.4.4 Supporting the Concept of a Region

An agent system supports a region by cooperating with other agent systems of the same authority and by supporting a region access point. The region access point is in charge of routing external travel requests to internal agent systems.

1.4.5 Finding a Mobile Agent

When an agent wants to communicate with another agent, it must be able to find the destination agent system to establish communication. The ability to locate a particular mobile agent is also important for agent management.

Because a mobile agent travels, an agent name must be unique across all agent systems within regions. Agent systems may provide a naming service based on agent names.

1.4.6 Ensuring a Secure Environment for Agent Operations

Because a mobile agent is a computer program that can travel among agent systems, a mobile agent is often compared to a virus. So, it is imperative for agent systems to identify and screen incoming agents. An agent system must protect resources including its operating system, file system, disks, CPU, memory, other agents, and access to local programs.

To ensure the safety of system resources, an agent system must identify and verify the authority that sent the agent. The agent system must also know what access the authority is allowed. The ability to identify the authority of an agent enables access control and agent authentication within an agent system.

Another aspect of security is confidentiality. For example, one agent traveling to meet another agent might want to keep both the occurrence of the meeting and the substance of the interaction confidential.

This section outlines security threats and countermeasures. It also discusses the required security services that the underlying communications infrastructure should provide.

Threats and Attacks

Agent systems may be vulnerable to security threats due to weaknesses in the communications infrastructure and programming languages. This document is mainly concerned with communications security threats, including:

- Denial of service, which is a reduction of the availability of an agent or agent system to legitimate users.
- Unauthorized access or use that occurs when an unauthorized person or computer program invokes operations of an agent or agent system.
- Unauthorized modification or corruption of data that occurs when an agent's or agent system's data are altered or destroyed, or false data is added.

An attacker can use a number of techniques to attack agent and agent system communications. Some possible attack techniques are:

- Spamming, which is flooding a service with illegitimate (or even legitimate) requests.
- Spoofing or masquerade, which is an agent or agent system falsifying its identity to get access to information and services.
- Trojan horse, which is an agent or agent system posing as a legitimate agent or agent system that can potentially receive private information from unsuspecting clients.
- Replay, which is recording and replaying a communications session. If a replay attack is not detected, and the repeated operations are cumulative, the attack can have a disastrous effect on the provided service.
- Eavesdropping, which is monitoring communications to obtain private information.

Strategies for Countering Threats and Attacks

To ensure that agents act responsibly, sets of rules are created and defined as security policies. These rules govern an agent's activities. The security and safety services that the underlying communications infrastructure and the programming language provide enforce the rules.

Both agents and agent systems can have security policies. The authority that the agent or agent system represents sets the policies. An agent or agent system can have multiple security policies. The particular policy to enforce is determined based on the authenticity of the communicating parties credentials, agent class, agent authority, and/or other factors.

Security policies contain rules for various purposes, including:

- Restricting or granting agent capabilities.

The affected capabilities could include creating new agents, traveling, or spending digital money (for example, Cybercash). Enforcing these rules is generally done using capability checks that are built into the agent programming language or runtime environment.

- Setting agent resource consumption limits.

  The region administrator can limit resources, including CPU usage, memory and disk consumption, the number of new agents created, and the number of network connections allowed. Generally, resource metering is built into the agent programming language or runtime environment to enforce resource limits.

- Restricting or granting access.

  The region administrator can control access to travel destinations, operations that an agent can invoke, and data that an agent can view, alter, or provide. Communications security mechanisms (for example, access control lists) or agent programming language features enforce access control.

Specifying the Level of Network Communications Security

When an agent invokes an operation or wishes to travel, the agent specifies its requirements for the quality of network communication security. These requirements include:

- Confidentiality.

  The agent may require the communications channel be secure from eavesdropping. The agent may also specify the strength of the encryption.

- Integrity.

  The agent may require integrity checks to detect any corruption or unauthorized modification of data during network communications.

- Authentication.

  The agent may require authentication of the destination agent system before it begins transmission. Communicating only with agent systems that it can authenticate helps an agent prevent unauthorized access to the information that it is carrying.

- Replay detection.

  The agent may require an agent system to use a replay detection algorithm to prevent duplication of the agent during a communications session.

Authentication

Usually, agent systems are easily authenticated where the agent system is co-located with the information that their authority uses to authenticate itself. Authentication services normally available in secure communications infrastructures include this functionality.

Agent systems use communication transport calls (e.g.RPC) to transfer agents between systems. To satisfy the destination agent system’s security policies, mutual authentication of agent systems may be required. Agent systems operate without human supervision. Therefore, both agent systems involved in the agent transfer must participate in the authentication process without human intervention (for example, without entering a password).

Agent authentication differs from agent system authentication. Agents cannot carry their encryption key with them when they travel (for example, private keys).

Instead, agent authentication uses authenticators. An authenticator is an algorithm that determines an agent's authenticity. An authenticator uses information such as the authenticity of the source agent system or launching client, the authorities of the agent and agent system involved, and possibly information about which authorities are trusted in order to authenticate an agent.

Authenticators, like agent systems, have types. Furthermore, authenticator types must be registered with a naming authority.

Authenticators are divided into two types: one-hop and multi-hop. It is currently possible to specify the behavior of and requirements for a one-hop authenticator. However, due to the limits of current security technology, specifying a multi-hop authenticator must be postponed.

A one-hop authenticator can authenticate an agent traveling one hop from its source agent system. For example, an agent of authority A is executing on a source agent system of Authority A, then migrates to a destination agent system of authority B. If destination agent system B can successfully authenticate source agent system A, the agent retains its source agent system A authenticity on destination agent system B. If destination agent system B cannot authenticate source agent system A, the agent is not defined as authenticated.

If an agent has a different authority than the source agent system, the agent transfer is considered a multi-hop operation. This specification does not address multi-hop authentication.

Countering Threats

This section presents several examples of how to use security policies to counter threats to the integrity of agent communications. The two general problems covered in these scenarios are denial of service and unauthorized access of data and services.

Suppose an agent executes code that attempts to consume all system resources. When resources are not available, services are denied to other agents and possibly to code running on the agent system host. To resist an attack like this, the region administrator can impose resource constraints on agents or code originating from untrusted sources.

In most cases, including CORBA, the level of trust assigned to an agent is partially a function of the agent's authority and whether that authority was authenticated. The trust level may also depend on digital signatures (generated strings or numbers that identify the author), or other techniques.

If a security method such as digital signatures is used, an agent can be trusted even if it arrives from an untrusted node. In such cases, the signed pieces can be trusted and only the pieces the untrusted node modified are suspect.

The communications infrastructure is responsible for authenticating the agent's authority. The agent system can supply any other safety services used in this scenario. In addition, the agent language can supply safety features that further enhance security.

Another possible attack on system resources can occur when an agent system is flooded with communications traffic. Usually, the lower-level communications equipment deals with this type of attack. An agent system might have no inherent defense against such an attack.

In an agent-based application, there may be an infinite number of techniques for gaining unauthorized access to data or services. The remainder of this section presents two possible threats to data and service integrity, and the defenses against them.

Suppose an attacker can monitor communications traffic that transports agents and decodes their state data. Once the state data is decoded, the attacker has access to any private information that the agent is carrying.

To counter this attack, an agent carrying sensitive information may demand confidentiality services as a condition for transport. If the level of protection required is not available, the agent transport should fail.

Suppose an attacker establishes an agent system that claims to operate on behalf of some trusted authority. Faking the identity of a trusted authority allows the attacker to receive agents that may be carrying information meant for only trusted parties. To counter this type of attack, an agent may demand that it only be transferred to agent systems that are authenticated.

Security Service Requirements

This section defines and describes the requirements for secure mobile agent communications, which are:

- Client authentication for remote agent creation
- Mutual authentication of agent systems
- Agent system access to authentication results and credentials
- Agent authentication and delegation
- Agent and agent system security policies
- Integrity, confidentiality, replay detection, and authentication

Authentication of Clients for Remote Agent Creation

Security services must provide for the authentication of non-agent system client applications. This authentication might be done using passwords or smart cards. Authenticating a client establishes the credentials of agents that the client launches. Client credentials also determine which security policy is used.

Mutual Authentication of Agent Systems

Agent systems, operating without human intervention, must be capable of authenticating each other. This authentication is accomplished by proving that the agent system is in possession of some secret information such as a private key. It may be acceptable for a human to enter a password at the time of the agent system's initialization to authorize the agent system to have access to the secret information.

Agent System Access to Authentication Results and Credentials

When agent communication takes place, the destination agent system must sample the credentials of both the agent and the source agent system, and verify their authenticity. An agent authenticator can use this information to authenticate an agent. The result of the authentication process determines which security policy to apply to the communications between the agent and the hosting agent system.

Agent Authentication and Delegation

If an agent is migrating to destination agent system, the agent's credentials must be transferred with the agent if the migration succeeds. The credentials may be weakened depending on the results of the authentication.

If the communication taking place is a remote method invocation, the client agent's credentials are passed along to the server agent for charging or auditing. When a client agent makes an RPC call, the client agent's credentials are made available to the server agent. If the server agent makes an RPC call on behalf of the client agent, the server agent should be able to pass the client agent's credentials.

An agent uses its thread of execution to take actions (such as making RPC calls) on its own initiative. When an agent takes such an action, the credentials associated with that action must be those of the agent so that the correct security policy is applied.

Agent and Agent System Security Policies

An agent should be able to control access to its methods. The agent or its associated agent system must both set and enforce the access controls. If the agent or agent system's access controls are both self-defined and self-enforced, the source agent's credentials must be available to the destination agent system, because this information is needed for access control decisions.

Alternatively, the agent or agent system set the access controls, then require the communications infrastructure to enforce them. For example, an agent could construct an access control list, then deliver it to the communications infrastructure for enforcement.

Integrity, Confidentiality, Replay Detection, and Authentication

For any communication, the requestor must be able to specify its integrity, confidentiality, replay detection, and authentication requirements. The communications infrastructure must honor these requirements, or return a failure indication to the requestor.

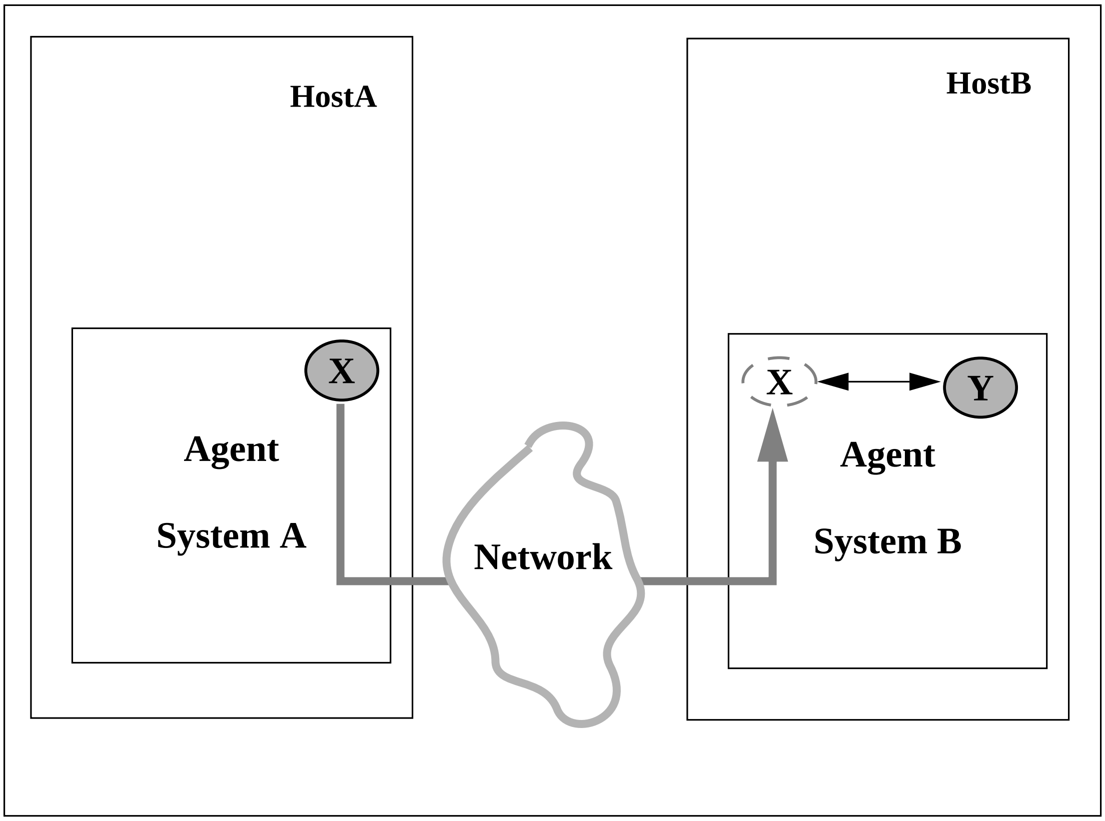


Figure 1-5 Agent transfer where both agents systems are of the same type

1.5 Agent System Interoperability Scenarios

Each agent system has a type (for example, Aglets), and can only create and support agents of that type. An agent of one type communicates with an agent of a different type by several cases involving agent system interoperability.

Case 1: Agent X wants to communicate with Agent Y. Agent X initially resides in Agent System A and Agent Y resides in Agent System B (see Figure 1-5). Because the communication involves a complex transaction, Agent X wants to transfer to the same host or network as Agent Y to take advantage of locality.

If Agent System A and Agent System B support the same agent profile, there is no interoperability problem to solve. Agent X simply requests a transfer to Agent System B, gets whatever information it needs, then returns to its original agent system (see Figure 1-5).

However, if Agent System A and Agent System B support different agent profiles, there are two ways for the agents to communicate. In one of the two cases local communication is possible, in the other case it is not.

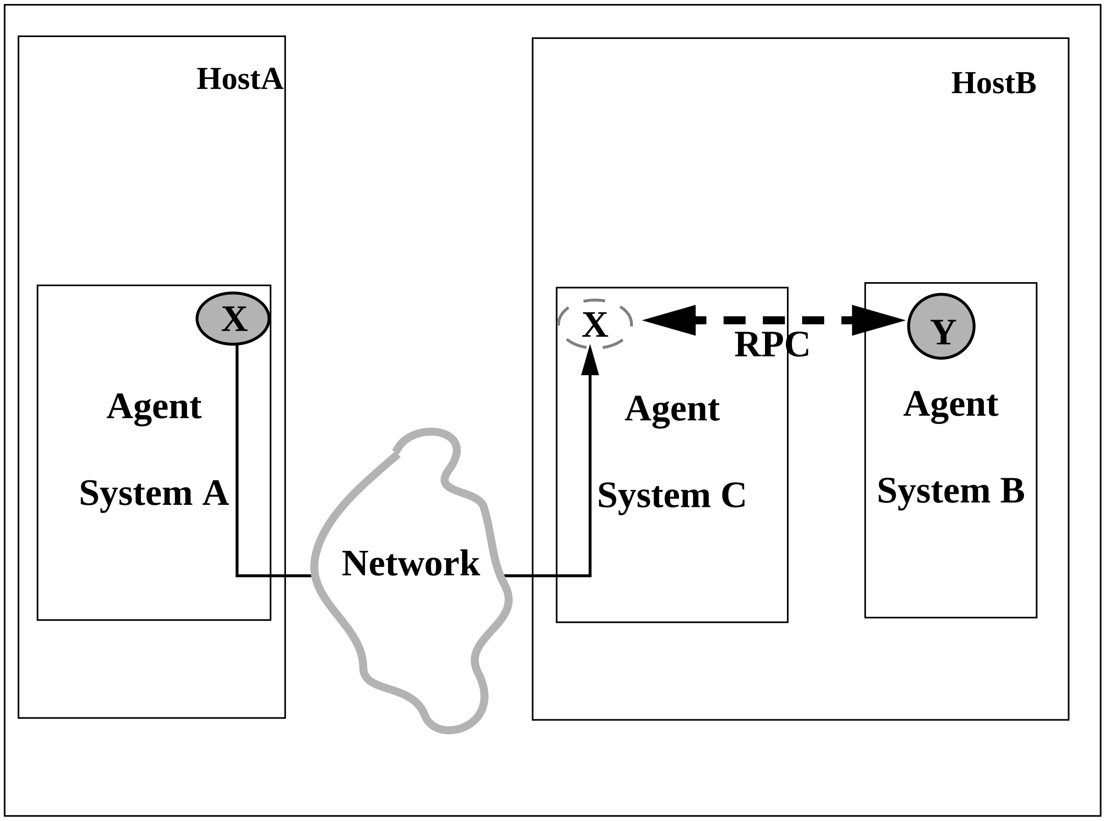


Figure 1-6 Agent transfer where the source and destination agent systems support the same agent profile

Case 2: Agent can take advantage of locality

In this case, a new agent system is added (Agent System C). Agent System A and Agent System C supports the same agent profile, and Agent System C is local (same host/network) to Agent Y (see Figure 1-6). Agent X discovers this fact via a call to Agent System B, then requests a transfer to Agent System C. Once Agent X travels to Agent System C, it can communicate locally with Agent Y via RPC, get the information it needs, then return to Agent System A.

Case 3: Agent cannot take advantage of locality

In this case, Agent X cannot find an agent system local to Agent System B that supports the same agent profile as Agent System A. So, it remains on HostA, and communicates with Agent Y across the network via RPC. Although it does not have the advantage of locality, communication can still occur. (see Figure 1-7)

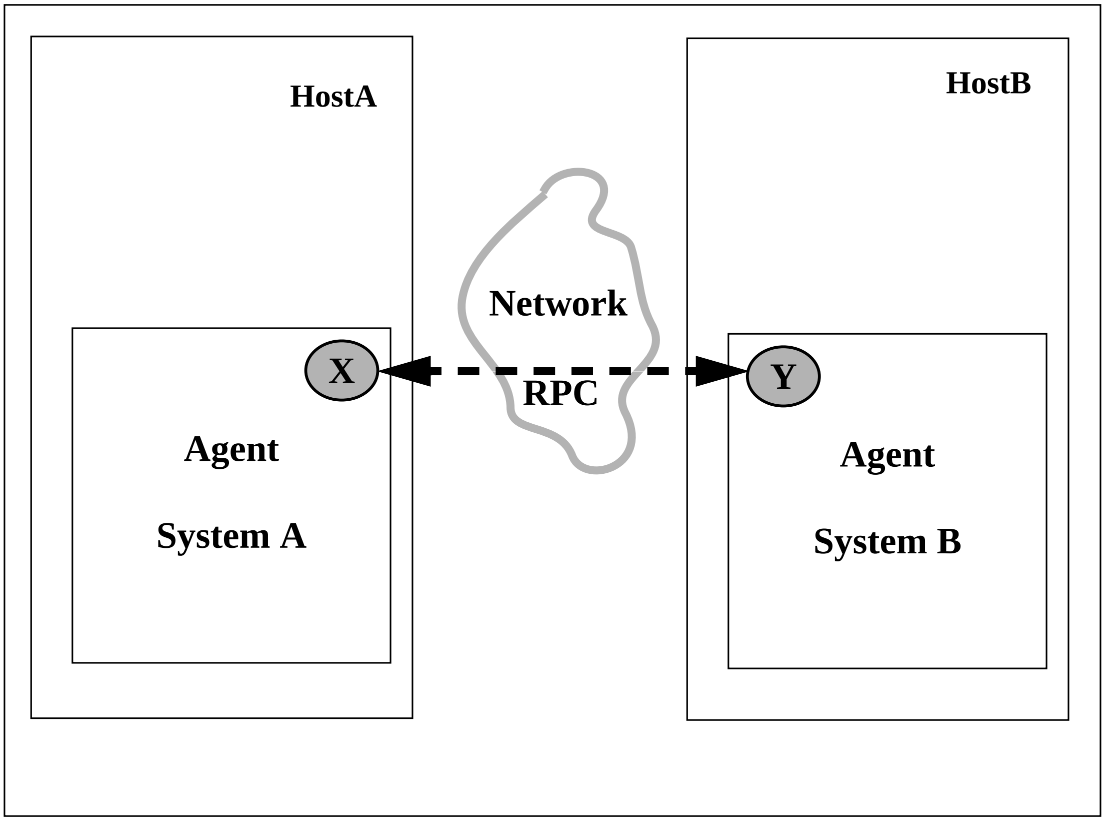


Figure 1-7 Agent unable to find compatible local agent system

CORBA Services 2

This chapter contains brief descriptions of the CORBA services that are related to mobile agent technology (see Figure 2-1):

- Naming Service
- Lifecycle Service
- Externalization Service
- Security Service

2.1 Naming Service

The CORBA Naming Service binds names to CORBA objects. The resulting name-to-object association is called a *name binding*, which is always related to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique. Naming contexts can be combined to a *naming graph*. This directed graph consists of nodes (represented by naming contexts) and labeled edges. A specific object can be addressed by a sequence of names (called a *compound name*) that builds one specific path in the naming graph.

Applications use the Naming Service to publish named objects, or to find an object given only the name. To obtain a reference to a naming service, an application typically bootstraps a reference to a naming context using the ORB::resolve_initial_references operation.

This MAF submission describes two CORBA object interfaces: MAFAgentSystem and MAFFinder. These objects may be published in the Naming Service if it is desired. It is not mandatory that the user does this, but it may offer some programming convenience. For example, an agent entering a region may use the Naming Service to get a reference to the MAFFinder.

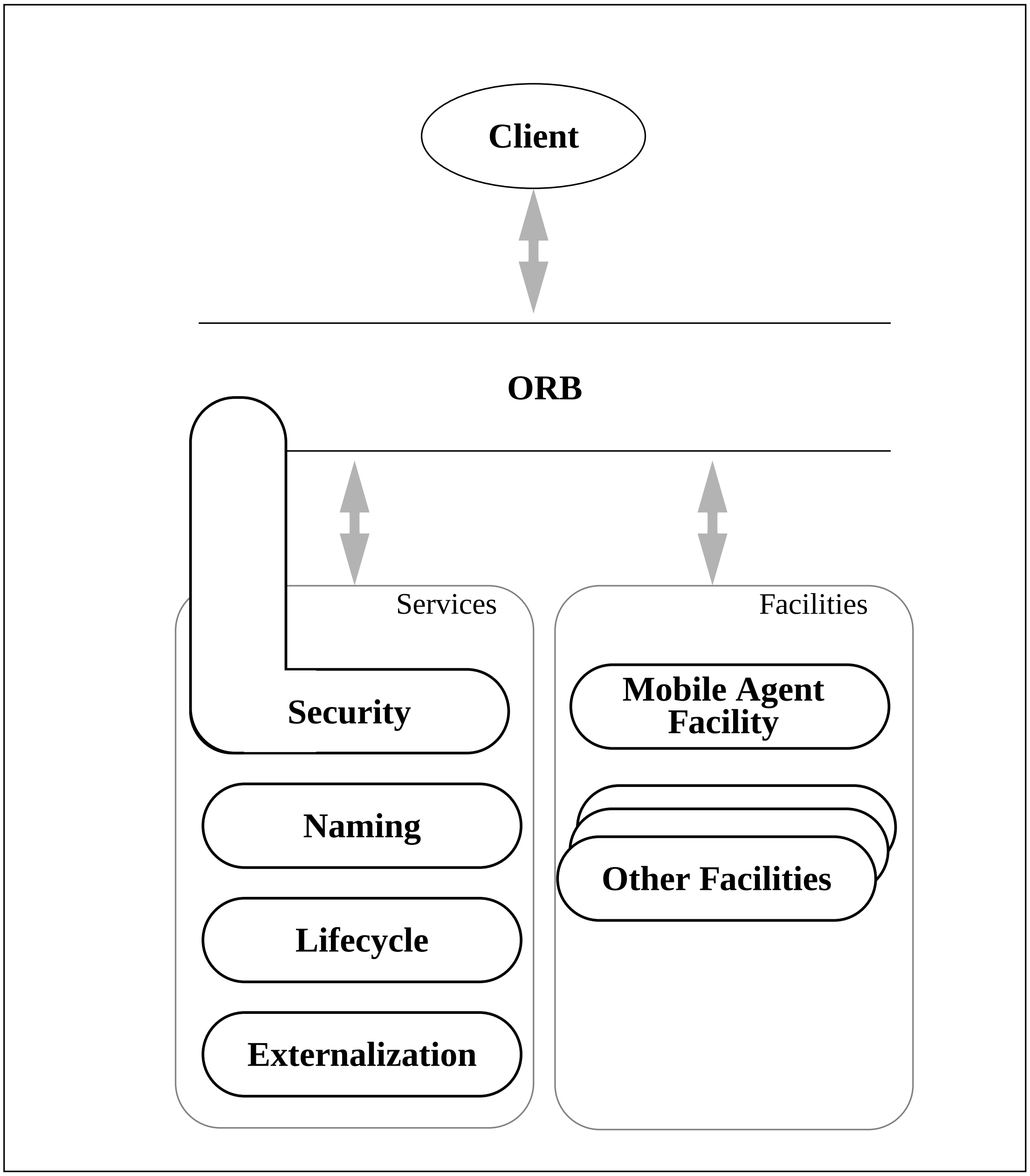


Figure 2-1 CORBA Services and Facilities

Agents that wish to act as CORBA objects may also choose to publish themselves using the Naming Service. Doing so gives applications a way to dynamically get object references to remote agents. Using this reference, an application can interact with the agent using CORBA RPC.

Stationary agents as well as mobile agents may publish themselves. Since a CORBA object reference (IOR) comprises, among others, the name of the host on which an object resides and the corresponding port number, a mobile agent gets a new IOR after each migration. In this case, the IOR that is kept by the accessing application becomes invalid. However, this problem can be solved in different ways. The following bullets show three possible solutions:

- The first solution is that the ORB itself is responsible for keeping the IOR of moving objects constant. The mapping of the original IOR to the actual IOR of the migrated agent is managed by a corresponding proxy object which is maintained by the ORB. Although this capability is described by CORBA (cf. CORBA 2.1, Chapter 12 “General Inter-ORB Protocol”), it is not a mandatory feature of an ORB. Thus, the MAF standard does not rely on this feature.

- The second solution is to update the name binding associated to the mobile agent after each migration, i.e. to supply the Naming Service with the actual agent IOR. This can be done by the agent systems which are involved in the migration process or by the migrating agent itself. In this way, the Naming Service maintains the actual IOR during the whole lifetime of the agent. If an application tries to access the agent after the agent has changed its location, the application retrieves an exception (e.g. *invalid object reference*). In this case the application contacts the Naming Service in order to get the new agent IOR.

- When a mobile agent migrates for the first time, the original instance remains at the home agent system and forwards each additional access to the migrated instance at the new location. In this way, the original IOR remains valid, and the clients accessing the agent need not care about tracking it. They still interact with the original instance, called proxy agent, which only exists to forward requests to the actual (migrating) agent.
  One disadvantage of this solution is that the proxy agent must be contacted by the migrating agent after each migration step in order to retrieve the new IOR to which each access request must be forwarded.
  Another disadvantage is that the home agent system must be accessible at any time. If the home agent system is terminated, the agent cannot be accessed anymore, since the actual IOR is only maintained by the proxy agent. This problem is essential, since it should be possible to create mobile agents on a device which is not permanently available (e.g. a laptop), to send the agents to another location and, after this, to separate the device from the network.

In order to uniquely identify MAF agent systems and agents, the following components are used:

- Authority: defines the person or organization which the agent or agent system represents

- Agent System Type: defines the type of an agent system. In case of an agent identification, this component represents the type of that agent system where the agent has been created.

- Identity: distinguishes agent systems or agents, respectively, which have the same authority and the same agent system type values.

The combination of these three components must be unique in the context of MAF. In the context of the CORBA Naming Service, each of the components is represented by one CosName.Name object.

The MAFFinder object is independent of specific authorities. The identification of such an object is managed by means of a single CosName.Name object corresponding to the CORBA Naming Service.

2.2 Lifecycle Service

The CORBA Life Cycle Service defines services and conventions for creating, deleting, copying and moving CORBA objects.

The CORBA objects defined by the MAF standard, i.e. MAFAgentSystem and MAFFinder, can be created and deleted using the Life Cycle Service. If desired, even copying or moving these objects is possible. This may be desired if the host on which an agent system is running must be shut down or separated from the network without disturbing the executing agents. Note that some management effort is necessary in this case, e.g. the address of the agent system that is maintained by the MAFFinder or Naming Service must be modified, and the residing agents must be supplied with the same runtime environment on the new host.

Mobile agents are active objects with the capability to move through the network. If an agent is represented as CORBA object, it is possible to use the Life Cycle Service for its creation, deletion, copying and migration. Since it is necessary to transfer the agent state, the Life Cycle Service must be combined with the CORBA Externalization Service.

As mentioned above, the Life Cycle Service can only be used for CORBA objects. Regarding the MAF standard, agents need not be CORBA objects. Thus, in order to provide a uniform interface for the creation, deletion and migration of CORBA-based and non-CORBA-based agents, new operations have been introduced for this purpose (cf. IDL specification of MAFAgentSystem). However, the *create_agent* and *terminate_agent* operations of the MAFAgentSystem interface can use the Life Cycle Service internally for CORBA-based agents.

2.3 Externalization Service

The CORBA Externalization Service provides a standardized mechanism for recording an object's state onto a data stream, and for restoring an object's state from a data stream. An agent system uses this service when it needs to serialize and deserialize an agent's state. However, the agent system implementor is free to choose any method (including non-CORBA mechanisms such as Java Object Serialization) for agent serialization and deserialization. (Note that in this section, we use serialization and externalization interchangeably. Deserialization and internalization are also synonymous.)

By using the Externalization Service to serialize an agent, the agent's state must be represented by a CORBA object that implements the Streamable interface. The agent system would also implement a MemoryStream object that has two purposes. 1) output an in-memory octet sequence when externalizing the agent. 2) read from an in-memory octet sequence when internalizing an agent.

A MemoryStreamFactory interface allows for the creation of MemoryStream objects. A suggested set of interface definitions is given below:

```
#include <CosExternalization.idl>
typedef sequence<octet> OctetString;
//MemoryStream externalizes objects to an in-memory
//octet sequence. After calling externalize() and
//flush(), the octet sequence representation may be
//accessed by calling get_octets().
interface MemoryStream : CosExternalization::Stream{
    OctetString get_octets();
};

//Use the MemoryStreamFactory to create a MemoryStream
//object. Call create() to make an empty MemoryStream
//for object externalization. Call create_from_octets()
//to make a MemoryStream that can internalize the objects
//from the supplied octet sequence.
interface MemoryStreamFactory {
    MemoryStream create();
    MemoryStream create_from_octets(OctetString octets);
};
```

Once an agent is externalized and the octet sequence is retrieved from the MemoryStream, the octet sequence can be passed to the remote agent system's receive_agent() operation to transmit the agent's state. The receiving agent system constructs a MemoryStream from the received octet sequence using the create_from_octets() operation. The receiving agent system then calls the MemoryStream's internalize() operation to reconstitute the agent's state.

2.4 Security Service

This section describes how CORBA (ORB) implementations may fulfill the agent security requirements discussed in the security section of "Common Conceptual Model" earlier in this specification. This section also describes how current CORBA security specifications fall short of agent security requirements in some areas.

Although CORBA security does not currently meet all the needs of mobile agent technology, the MAF implementation must use available CORBA security to satisfy its security needs. Future versions of CORBA security should address these issues.

The security capabilities of current CORBA implementations can be categorized as follows:

1. No security services. The implementation includes neither proprietary nor standardized security interfaces. This type of implementation is limited to secure environments (either physically or firewall protected from unauthorized access), or to applications that contain no data or services worth protecting. Usually, Intranet applications provide no security services.

2. Proprietary security services. The implementation includes a vendor-defined set of security capabilities such as authentication and access control. These services may be transparent to the application, or may be accessed via vendor-defined interfaces. Note that these services do not involve the ORB, and therefore do not provide an acceptable level of safety.

3. Conforming implementation of CORBA security services (refer to the CORBA Security Services Specification and the Common Secure Interoperability Specification (CSI) for more information about CORBA-defined security services). The implementation includes security services that conform to CSI level 0, 1 or 2 as defined in the CSI document, and interfaces that are defined in the CORBA Security Services Specification.

The security requirements for agents and agent systems in CORBA are:

- Agent naming
- Client authentication for remote agent creation
- Mutual authentication of agent systems
- Agent system access to authentication results and credentials
- Agent authentication and delegation
- Agent and agent system security policies
- Integrity, confidentiality, replay detection, and authentication

This section of the specification describes agent security in terms of category 1 and 3 of CORBA implementations. The analysis given here could also be performed for category 2 of CORBA implementations on a case by case basis.

2.4.1 Agent Naming

The destination agent system must identify the principal on whose behalf an agent is acting. This is true even when that principal is not authenticated, because certain applications may find it acceptable to use application-defined heuristics to evaluate authenticity.

An agent system can provide the following information to an authorized user about an agent that it is hosting (CORBA security uses the term principal instead of authority):

- The agent's name (principal and identity)
- Whether or not the principal has been authenticated (authenticity)
- The authenticator (algorithm) used to evaluate the agent’s authenticity

Secure ORBs exchange security information about principals when remote operations are invoked. This information is available to an application, such as an agent system, as a Credential object. If an ORB does not support security services, however, or a principal is not authenticated, the principal identity information is not available (if the Credential is available, the only identity will be the Public identity).

It is necessary for agent systems to exchange principal information when agents are transferred. The information in the Credential, if available, may be used to evaluate the authenticity of the information exchanged. If the Credential is not available, the agents authenticity is automatically false.

2.4.2 Client Authentication for Remote Agent Creation

CORBA security services offer client authentication services via the PrincipalAuthenticator object. Note that these services are not agent specific.

The client invokes the authenticate operation to establish its credentials. When the client makes a request to an agent system to create an agent, it makes these credentials, which were obtained via PrincipalAuthenticator object, available to the destination agent system. The principal for the new agent is then obtained via this credentials. The agent system uses this information to find and apply the appropriate security policies.

A non-secure ORB does not provide client authentication. If a client creates an agent in such an environment, the client may supply a name for the agent, but the agent will be marked as “not authenticated”.

2.4.3 Mutual Authentication of Agent Systems

CORBA security services allow administrators to require the mutual authentication of agent systems by setting the association options for agent systems. Specifically, both the EstablishTrustInClient and EstablishTrustInTarget association options are required for agent systems.

Both the source and destination agent systems transfer credentials before an agent transfer occurs. This transfer makes it possible to apply security policy before transferring the agent. This policy protects against transferring agents to illegitimate agent systems, and protects against agent systems giving access to illegitimate agents.

A non-secure ORB does not provide mutual authentication of agent systems. An agent that is initially marked as authenticated is marked as “not authenticated” if the agent visits an agent system that cannot be authenticated.

2.4.4 Access to Authentication Results and Credentials

At the destination end of an agent transfer, CORBA security services provide access to the credentials of the source via the SecureCurrent interface. The get_credentials operation may be used to obtain a reference to a Credentials object. The Credentials object includes the sender’s principal if the sender was authenticated. For agent transfer, both sender and receiver are agent systems. The receiver of an agent transfer request may evaluate the sender's credentials to determine the identity and authenticity of the sender.

On the other hand, if an agent invokes operations on CORBA objects, the agent needs to have the credentials for its principal for secure invocations. This is true even if the agent is defined as non CORBA object. In this case, the credentials object of the agent should be available at the destination. Therefore, the credentials of both the agent systems and the agents must be available at the destination to built a secure agent system.

If a secure ORB supports CSI level 2 with composite delegation, credentials of both the agent's principal and the sender agent system's principal can be made available at the receiver side. These credentials are obtained using the SecurityLevel2:Current interface. Then the agent's credentials are used for securing subsequent invocations the agent performs. Refer to Section 2.4.5, "Agent Authentication and Delegation" for more information.

If composite delegation is not supported, it is not possible to make both credentials available at the new agent system. If a secure ORB supports only the Security Functionality level 1, an agent system cannot control which credentials are used for a transfer request the agent system issues and the security policy defined by the security authority governs it.

It is possible for a sender agent system to set and use the agent system's credentials for the agent transfer if a secure ORB supports the Security Functionality level 2. The agent system may use SecurityLevel2:Current the interface to set its credentials or may use the override_default_credentials on the reference of the target agent system.

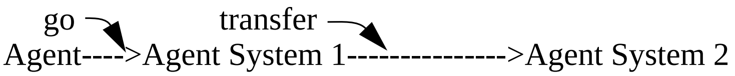
If the agent system's credentials are used for the agent transfer, the destination agent system can evaluate the sender agent system's principal to determine the identity and authenticity of the sender. However, the invocation credentials for the agent may become those of the agent system that is hosting the agent. This makes it much more difficult to secure operation invocations the agent performs, because object invocations by any agent appear to have the authority of the agent system's principal.

For an agent system to secure such invocations without having the CSI level 2, it may choose to use the agent's credentials for the agent transfer. The receiver agent system can then use that credentials for the agent's secure invocations. In this case, however, the sender's credentials are not available and the receiver cannot evaluate the sender agent system's principal.

Note that the secure agent system can be built on top of CSI level 0 or 1 if an agent does not invoke operations on CORBA objects. In a non-secure ORB, all agent transfers and agent operation invocations are anonymous. The only identifying information available is the unauthenticated principal value that an agent system may include during an agent transfer. The ORB does not transfer or support access to credentials.

2.4.5 Agent Authentication and Delegation

When possible, it is desirable that secure ORB implementations propagate the agent's credentials along with the agent as it moves between agent systems. This may only be possible using composite delegation, which involves both parties in the transfer request, then propagates the credentials of the agent and the sending agent system (see the following diagram).



The diagram shows the call flow beginning with an agent making a request to its hosting agent system (Agent System 1) for a transfer to Agent System 2. When composite delegation is supported, the credentials of both the agent and Agent System 1 are available to Agent System 2.

Upon receiving an agent's credentials, the receiving agent system should establish the agent's credentials as the invocation credentials of the agent. By doing so, any operations invoked by the agent are subject to the policies associated with the agent's principal. This approach also ensures the continued propagation of the agent's credentials when the agent makes other transfers.

If an agent system receives an agent from an untrusted agent system, the agent system may choose to weaken the agent's credentials. For example, the agent system may wish to treat the agent as unauthenticated.

The propagation of both agent’s credentials and agent system’s credentials is only possible with composite delegation, which is only available with ORB implementations that conform to CSI level 2. Furthermore, it is not known whether ORB implementations will support delegation of credentials to application-created threads of execution. Delegation of credentials is needed to identify an agent’s principals when an agent invokes a method on CORBA objects.

In CSI level 0 and 1 implementations, only one of the credentials of the agent or the agent systems can be transmitted. If mutual authentication between agent systems is not required (such as in case of the trusted environment), the agent’s credentials may be propagated to the destination agent system in lieu of the agent system’s credentials. In non-secure ORB implementations, an agent’s credentials are not propagated between agent systems; thus, there are no available credentials for agents that travel.

2.4.6 Agent and Agent System Defined Security Policies

Any CORBA object implementation may refuse to service a request. Secure ORB implementations (CSI levels 0,1, or 2) can provide the object implementation with the credentials of the requestor, allowing object implementations to make their own access decisions. Typically, when a CORBA object implementation throws exception CORBA::NO_PERMISSION of a type that indicates that a security violation was attempted and refused.

Object implementations based on non-secure ORBs do not have the requestor's credentials available. However, these implementations may refuse to service a request based on other criteria, such as the values of the request parameters.

2.4.7 Security Features

Secure ORB implementations allow applications to specify the quality of security service when they invoke operations. To specify the security level, set the security features of the invoker's credentials, or set the quality of protection in an object reference.

Security features that are set via the invoker's credentials include:

- Integrity
- Confidentiality
- Replay detection
- Misordering detection
- Target authentication (establish trust)

Security features that are set via the quality of protection in an object reference include:

- Integrity
- Confidentiality

MAF IDL 3

3.1 Overview

The Mobile Agent Facility (MAF) is a collection of definitions and interfaces that provide an interoperable interface for mobile agent systems. MAF is as simple and generic as possible to allow for future advances in mobile agent systems.

Reasons for standardizing certain areas of mobile agent technology have been described in the section Section 1.5, “Agent System Interoperability Scenarios“” earlier in this specification. Please refer to that section for a discussion of the advantages and design goals of MAF.

The MAF module contains two interfaces:

- MAFAgentSystem interface
- MAFFinder interface

The MAFAgentSystem interface defines agent operations, including receive, create, suspend, and terminate. The MAFFinder interface defines operations for registering, unregistering, and locating agents, places, and agent systems.

The interfaces have been defined at the agent system level rather than at the agent level to address interoperability concerns. Both agent systems and agents may be, but are not necessarily, CORBA objects. However, agents live in agent systems, and therefore the implementation of an agent depends upon the implementation of the agent system that creates it.

Currently, several agent systems have been built, including IBM’s Aglets, General Magic’s Telescript, and Dartmouth College’s AgentTcl. These agent systems differ from each other. For example, they use different languages, encoding/serialization methods, and authentication mechanisms.

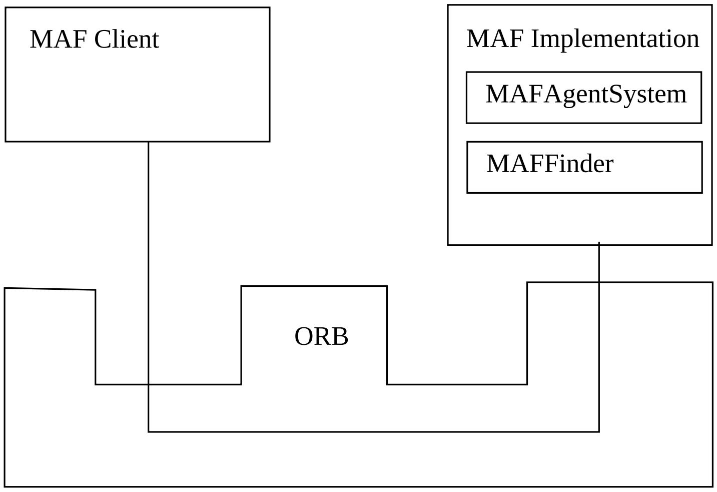


Figure 3-1 The relationship Between MAF and an ORB

Because agents travel only from one agent system to another that supports the same agent profile, there is no need to unify the agent interface. Instead, the agent management operations defined in MAF, such as suspend, unregister, and terminate agent are standardized. These operations form a basic set that is sufficient for agent inter-system travel.

The CORBA services are designed for static objects. When CORBA naming services, for example, are applied to mobile agents, they may not handle all cases well. Therefore, a MAFFinder interface is also declared here. The MAFFinder functions as an interface of a dynamic name and location database of agents, places, and agent systems.

Before each of these interfaces is discussed in detail, some predefined structures and definitions used in these modules will be discussed. This discussion is in the two sections titled Section 3.3, “Name, Class Name, and Location“, and “Section 3.4, “OMG Naming Authority Identifiers“.

3.2 *The MAFFinder*

The MAFFinder is a naming service. It may be shared among regions. However, for simplicity of definition, there is one MAFFinder for each region. Figure 3-2 shows the two region scenario used in describing the MAFFinder concepts.

Before a client can request the MAFFinder to find an object, the client must obtain the object reference to the MAFFinder. To get the object reference, the client uses either the CORBA Naming Service or the method AgentSystem.get_MAFFinder().

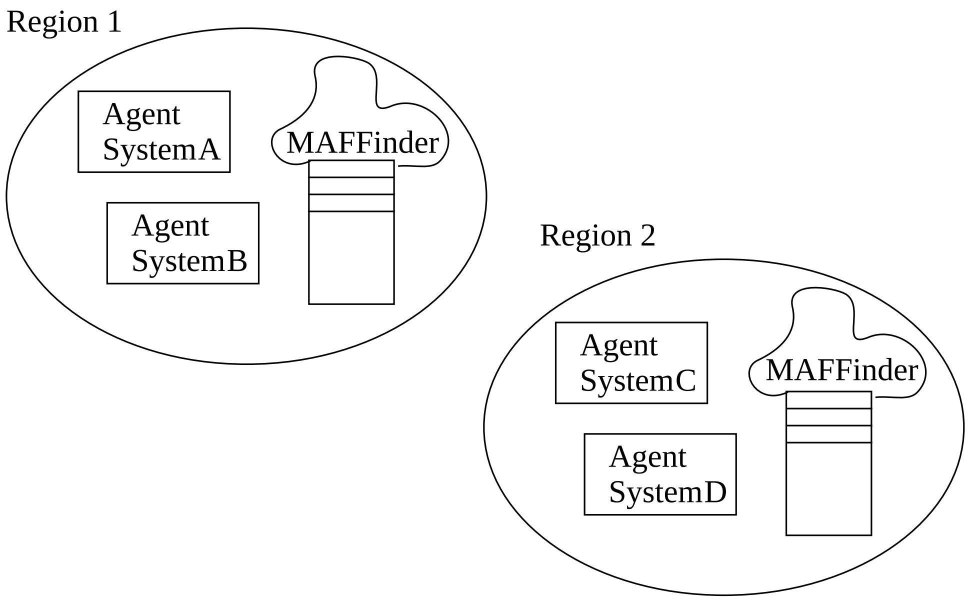


Figure 3-2 Two Region Scenario

3.3 Name, Class Name, and Location

```
typedef short AgentSystemType;

typedef sequence<octet> OctetString;
struct ClassName{
      string name;
      OctetString discriminator;
};
typedef sequence<ClassName> ClassNameList;
typedef sequence<OctetString> OctetStrings;
typedef OctetString Authority;
typedef OctetString Identity;

struct Name{
      Authority               authority;
      Identity                identity;
      AgentSystemType         agent_system_type;
};
typedef string Location;
```

3.3.1 Name

In the MAF module, Name is defined as a structure that consists of three attributes: authority, identity, and agent_system_type. These attributes create a globally-unique name for an agent or agent system.

When Name is an agent name, the agent_system_type is the type of agent system that generated the identity of the agent. When Name is an agent system name, the agent_system_type is the agent system's type.

Authority defines the person or organization the agent or agent system represents. CORBA uses the term principal instead of authority. The authority of the agent must be equivalent to the principal of the agent's credentials if the CORBA security is used.

Agent systems of different types may use different mechanisms to generate identities. Therefore, it is possible that two agent systems of different types might generate the same authority and identifier. In addition, the responsibility for naming an agent may also differ for each agent system types. The client may be responsible for naming in some agent system, while the agent system must generate a name for the agent in others. MAF specification allows these two approaches.The agent_system_type distinguishes one agent system from another with the same authority and identifier.

The Name structure defines the syntax for an agent or agent system name.

3.3.2 Class Name

The ClassName structure defines the syntax for a class name. A class name has a human-readable name and an octet string that ensures that the class name is unique within the scope.

This specification does not provide any mechanism to make class names globally unique. So, an agent system should not assume that class names are globally unique. Therefore, MAF implementors are responsible for ensuring that class names are unique within the scope of the source agent system for either a receive_agent() or create_agent() call.

Figure 3-3 illustrates the minimum requirement for class name uniqueness, which is uniqueness within an agent system.

When Agent System C requests ClassOne from Agent System A, ClassOne should be unique within the scope of Agent System A. Similarly, when Agent System C requests ClassOne from Agent System B, this ClassOne should be unique within the scope of Agent System B.

Agent System C must distinguish between the two versions of ClassOne. This is necessary, for example, if Agent System D needs ClassOne to create an instance of an Agent System A type of agent.

Suppose Agent System C wants to create an agent on Agent System D. If a ClassOne is involved in this creation, Agent System C uses the class_names parameter in Create_Agent to specify for Agent System D which ClassOne is necessary. Once Agent System D receives the class, it can rename it.

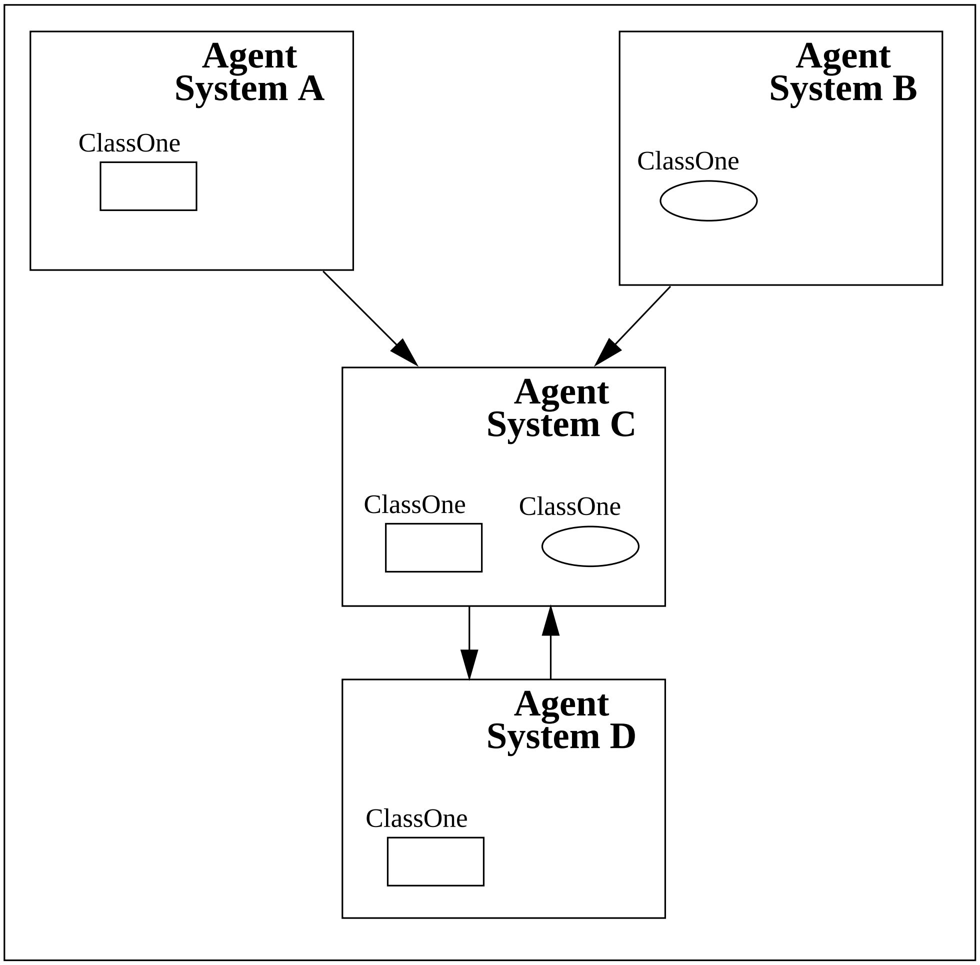


Figure 3-3 Class name uniqueness within an agent system

Note that the difference between names for the same version of ClassOne on the two agent systems can cause an unnecessary class transfer. For example, if Agent System D later attempts to transfer an agent that uses the Agent System A version of ClassOne to Agent System C, Agent System C might not recognize that the ClassOne specified in the class list for the call is the same as the class A:ClassOne that it already has. It is not within the scope of this submission to address this situation, however.

If the region administrators of the communicating(source/destination) agent systems agree on a globally-unique class naming scheme, the problem of duplicate names for the same class can be avoided. For example, if class names were globally unique in Figure 3-3, Agent System C would not ever encounter two classes that had the same name.

3.3.3 Location

In the MAFFinder interface, Location specifies the path to an agent system based on the name of an agent system, agent, or place. For example, when MAFFinder.lookup_agent() is called using an agent name, a Location specifying the agent system that contains the agent is returned.

Once the client gets the Location of an agent system, it must convert the Location (a String) to the object reference of the agent system to invoke the operations offered.

The Location String is in one of two forms:

- a URI containing a CORBA name
- a URL containing an Internet address

The advantage of using the CORBA Naming Service is that it is protocol independent. The advantage of using an internet address is that it is better suited to mobile agents and the Internet.

To determine which format the Location is in, the client parses the string up to the first colon (:). If the characters preceding the colon are “CosNaming”, the string is a CORBA name. If the characters preceding the colon are “mafiiop”, the string is an Internet address.

COSNaming Location String Format

When the Location is in a CORBA name format, the client must convert the URI containing the CORBA name to the syntax of a CosNaming.Name (refer to RFC 1630, “Universal Resource Identifiers in WWW” for URI format details). Once the Location is converted to a CosNaming.Name, the client uses the CosNaming.Name as the key for a search that returns the agent system object reference.

The format of a CosNaming.Name is an ordered sequence of components. Each component consists of two attributes: the identifier and the kind, which are both strings.

The location for an agent system or a place can be written in URI form using the following syntax (refer to RFC 1630 for a definition of xpalphas):

mafuri	:=scheme”:”location
scheme	:=“CosNaming”
location	:=components | “/”location
components	:=component | component”/”components
component	:=id”!”kind
id	:=xpalphas
kind	:=xpalphas

Conversion of a CosNaming URI to a CosNaming.Name is a straightforward mapping from one to the other. In a CosNaming URI, the components are separated with slashes (/), and the identifier and kind attributes of each component are separated with an exclamation mark (!).

For example, the Location containing URI:

CosNaming:/user!domain/user_name!u3

can be converted to the following CosNaming.Name.

{{“user”, “domain”}, {“u3”, “user_name”}}

Internet-Specific (MAFIIOP) Location Conversion Method

If IIOP is used, an IOR for an agent system in another domain could be constructed directly from the location information. The requirements for an IIOP IOR are host name, port number, and an object key (an octet string defined in the CORBA Interoperation description of IIOP IOR). The host and port number might be expressed as part of the location information, and the object key for the agent system can be a string value (for example, “AgentSystem1” represented as an octet string).

Note that such references may not be actual object references, since CORBA objects may migrate and thus change their IOR. The IIOP redirection capability is used to map a received reference to an actual reference. The client’s ORB caches the corrected version of the reference. The requesting client object is oblivious to the redirection. This mechanism works for getting a reference to any CORBA object, and is not specific to MAF objects. However, regarding the CORBA specification, the redirection capability is not a mandatory ORB capability. Therefore, clients also must be able to get actual IORs, e.g. by contacting a Naming Service which is continuously supplied with the actual IORs of mobile objects.

The location for an agent system or a place can be written in URL form using the following syntax (refer to RFC1738, “Uniform Resource Locators”, for definitions of hostname, hostnumber, digit, and uchar).

```
mafurl          := scheme “:” location
scheme          := “mafiiop”
location        := “//” [hostport “/”] agentsystem [“/” place]*
hostport        := host “:” port
host            := hostname | hostnumber
port            := digit+
agentsystem     := uchar+
place           := uchar+
components      := component [“&” components]
component       := tagname “=” tagvalue
tagname         :=   “TAG_ORB_TYPE” |
                     “TAG_CODE_SETS” |
                     “TAG_SEC_NAME” |
                     “TAG_ASSOCIATION_OPTIONS” |
                     “TAG_GENERIC_SEC_MECH”
tagvalue        := uchar+
```

URLs of this type can define locations accessible via IP-based networks such as the Internet.

Even though a place is addressable via this scheme, it is not a requirement that places be first class CORBA objects. To get a reference to an agent system, a client manufactures an IOR using the host, port, and agent system components of the URL. If there is a place component (a path separated by one or more slashes), then it is passed as an argument to any agent system operations that may require it. If there are components (equations separated by "&"), then it denotes tagged components included an IIOP version 1.1 IOR. A tagname represents a tag of the component, and a tagvalue represents a value of the component.

Because locations can be written as strings, there need not be a special data type for them. The URL format given above can be used for defining locations accessible via the Internet or an Intranet.

Additional Location Conversion Method

For non-IP networks that do not use the CORBA Naming Service, other URIs (refer to RFC1630, "Universal Resource Identifiers in WWW") could be developed. Those URIs are distinguished from the URL defined above by choosing different scheme tags. The location specification and how it can be mapped to an IOR will be defined.

3.4 OMG Naming Authority Identifiers

The identifiers assigned to parameters such as agent system type and authenticator should be unique across all implementations of MAF. Therefore, it is necessary that some group assign and maintain these parameters. We propose that the OMG be the naming authority for the MAF parameters that require global uniqueness. OMG should assign values and manage the parameters of the following definitions:

```
typedef short LanguageID;
typedef short AgentSystemType;
typedef short Authenticator;
typedef short SerializationID;
typedef sequence<SerializationID> SerializationIDList;
```

These parameters are used in the following definitions:

```
typedef any Property;
typedef sequence<Property>                    PropertyList;

struct LanguageMap {
        LanguageID                  language_id;
        SerializationIDList         serializations;
};
typedef sequence<LanguageMap>                 LanguageMapList;

struct AgentSystemInfo {
                Name                        system_name;
                AgentSystemType             system_type;
                LanguageMapList             language_maps;
                string                      system_description;
                short                       major_version;
                short                       minor_version;
                PropertyList                properties;
};

struct AuthInfo {  //authentication information
        boolean is_auth;
        Authenticator authenticator;
};


struct AgentProfile {
        LanguageID                  language_id;
        AgentSystemType             agent_system_type;
        string                      agent_system_description;
        short                       major_version;
        short                       minor_version;
        SerializationID             serialization;
        PropertyList                properties;
};
```

This specification does not dictate which agent system types, languages, serialization mechanisms, and authentication methods must be used to accommodate new systems. The OMG naming authority should begin with these initial values:

- Languages: Java, Tcl, Scheme, and Perl
- Agent system type: Aglets, MOA, and AgentTcl
- Authenticator type: none, one-hop authentication
- Serialization methods: Java Object Serialization

The property names specified in the Property structure should also be managed by a naming authority. However, this is a future task, because there are currently no defined property names in the MAF specification.

Agents and agent systems provides application specific properties. A client may specify these properties e.g. in order to restrict the scope of a search operation while looking for a specific agent or agent system using the corresponding lookup method of the MAFFinder interface (see MAFFinder Interface). In order to specify a property, a client must support the application specific format of the value component of the property. The semantics and syntax of the value are identified by the name component of the property.

OMG has agreed to become the naming authority for mobile agent technology. Having a naming authority benefits the interoperability and unambiguous exchange of information between different MAF applications.

3.5 MAFAgentSystem Interface

The MAFAgentSystem interface defines methods and objects that support agent management tasks such as fetching an agent system name and receiving an agent. These methods and objects provide a basic set of operations for agent transfer.

```
interface MAFAgentSystem {

    Name    create_agent(
            in Name                 agent_name,
            in AgentProfile         agent_profile,
            in OctetString          agent,
            in string               place_name,
            in Arguments            arguments,
            in ClassNameList        class_names,
            in string               code_base,
            in MAFAgentSystem       class_provider) raises (ClassUnknown,
                ArgumentInvalid, DeserializationFailed,
                MAFExtendedException);

    OctetStrings fetch_class(in ClassNameList class_name_list, in string
        code_base, in AgentProfile agent_profile) raises (ClassUnknown,
        MAFExtendedException);
    Location find_nearby_agent_system_of_profile (in AgentProfile profile)
        raises (EntryNotFound);
    AgentStatus get_agent_status(in Name agent_name) raises
        (AgentNotFound);
    AgentSystemInfo         get_agent_system_info();
    AuthInfo    get_authinfo(in Name agent_name) raises (AgentNotFound);
    MAFFinder get_MAFFinder() raises (FinderNotFound);
    NameList list_all_agents();
    NameList list_all_agents_of_authority(in Authority authority);
    Locations list_all_places();

    void    receive_agent(
            in Name                 agent_name,
            in AgentProfile         agent_profile,
            in OctetString          agent,
            in string               place_name,
            in ClassNameList        class_names,
            in string               code_base,
            in MAFAgentSystem       agent_sender) raises (
                ClassUnknown, DeserializationFailed, MAFExtendedException);

    void resume_agent(in Name agent_name) raises (AgentNotFound,
        ResumeFailed, AgentIsRunning);
    void suspend_agent(in Name agent_name) raises (AgentNotFound,
        SuspendFailed, AgentIsSuspended);
    void terminate_agent(in Name agent_name) raises (AgentNotFound,
        TerminateFailed);
    void terminate_agent_system() raises (TerminationFailed);
    };
```

3.5.1 create_agent()

An agent system performs the create_agent operation to create an agent according to a remote client’s request. The actual name of the created agent is returned. This can be same as the name given as the parameter if the client is responsible for naming.

Syntax

```
Name create_agent(in Name agent_name, in AgentProfile agent_profile,
    in OctetString agent, in string place_name,
    in Arguments arguments, in ClassNameList
    class_names, in string code_base, in MAFAgentSystem class_provider)
    raises (ClassUnknown, ArgumentInvalid, DeserializationFailed,
    MAFExtendedException);
```

Parameters

agent_name — Name of the new agent. The agent’s authority is the client’s authority. Its agent_system_type is either the client’s agent_system_type, if the client is an agent system, or it is NonAgentSystem (value 0, refer to Appendix B, “Assigned Numbers“, for more information) if the client is not an agent system. The identity must be provided if the client is responsible for naming the new agent. If the agent system has the responsibility, the identity field is ignored. The actual name for the new agent is given as the return value.

agent_profile — Contains information about the agent, such as the agent system type that created it, and the method used to serialize it for transfer. Based on agent profile and profile of the target system, the latter can discover if agent requirements and system support are “similar” enough in order to accept agent at the site.

“Similar” is defined and interpreted by the system that accepts the agent. “Similar” can be defined by the type of the manufacturer. E.g. in certain cases there might not be interoperability between certain manufacturers, while at the same time there is interoperability with other manufacturers. For example, it is possible to envision that Aglets can interoperate with Odyssey and Magenta with MOA. “Similar” can also be defined by the versions of the system. Agent might have been created at the agent system of the same manufacturer but with an obsolete or incompatible version. “Non-similar” is certainly implied for different languages.

agent — This parameter is opaque, and can contain anything that the sender needs to convey to the remote agent system that is not covered in the other parameters to this method. This parameter contains information that is unique to agents of a particular agent profile. The agent system creating the agent must be able to decode the meaning of the information in this parameter.

The kinds of information that can be in this parameter include, but are not restricted to the agent class definition, the agent’s states, and definitions of some or all the classes needed to instantiate the agent at the remote agent system.

place_name — Name of the place where the agent will reside. If this parameter is not specified, the agent system creates the agent in a default place within the system.

arguments — This parameter specifies the arguments for the agent constructor.

class_names — List containing the name of each class necessary to instantiate the agent. Note that the class list is optional, which means it can be empty. Because CORBA does not accept null as a passed-in argument for a parameter specified as classNameList, the classNameList with the name field set to an empty string must be used to indicate the case where no class name is required. MAF implementors can provide a special null class name for convenience.

The list may or may not be necessary depending on the class transfer mechanism chosen. Refer to “Class Transfer“ in Section 1.3.1, “Remote Agent Creation“ earlier in this specification for a discussion of the various mechanisms and their requirements.

Classes listed here may or may not need to be transferred depending on whether the agent system caches classes. In fact, the receiving agent system must decide whether fetch_class() is needed.

code_base

Reference to the code base containing the necessary class definitions. The syntax of this parameter can vary between agent system types without affecting interoperability. It is returned to the requesting client via fetch_class(), if necessary. Therefore, only the class provider needs to know the syntax of this string if the source of the class definitions is an agent system. It is possible for the destination agent system to retrieve the class definitions directly by using the codebase information.

class_provider

Reference to the client source that provides the necessary class definitions. Note that the class_provider parameter should be void if the classes are provided by non agent system.

Exceptions

ClassUnknown

Failed to find class definition.

ArgumentInvalid

Arguments passed did not match any agent constructor’s signature.

DeserializationFailed

The agent system could not instantiate the agent because it could not decode the agent OctetString.

MAFExtendedException

This is a generic exception. Use it only if no other exception applies to the error condition.

Usage Notes

Even a remote client without agent capabilities can create an agent using this method. Because the client is not required to have agent capabilities, it can minimize its footprint. The size of the footprint is very important for a client in a handheld computer such as a Pilot or a MagicLink.

The destination agent system is free to queue the agent or route it to another agent system within its region. This is an implementation detail that the specification allows but does not mandate.

3.5.2 fetch_class()

The method fetch_class() returns definitions of one or more classes. In case of non object oriented agent system, fetch_class() method is used for fetching code.

Syntax

```
OctetStrings fetch_class(in ClassNameList class_name_list, in string
    code_base, in AgentProfile agent_profile) raises (ClassUnknown,
    MAFExtendedException);
```

Parameters

class_name_list — Names of the class definitions requested.

code_base — Reference to the code base containing the necessary class definitions. The syntax of this parameter can vary between agent system types without affecting interoperability. The client provides the code_base in create_agent() or receive_agent(). It is returned to the client via fetch_class(), if necessary. Therefore, only the class provider needs to know the syntax of this string.

agent_profile — Contains information about the language and serialization method used for the current context of create_agent() or receive_agent().

Exceptions

ClassUnknown — Failed to find class definition.

MAFExtendedException — This is a generic exception. Use it only if no other exception applies to the error condition.

Usage Notes

Use this method to retrieve classes from the specified code base and client. The requested agent system can know whether it is the class provider or not by examining the given codebase. If not, the requested agent system may return the cached classes if any, or may route this request to another agent system. This is an implementation detail that the specification allows but does not mandate.

3.5.3 *find_nearby_agent_system_of_profile()*

The find_nearby_agent_system_of_profile() method requests the MAFFinder to find a nearby agent system that can execute the agent that the client wants to send. This is an interface on the MAFAgentSystem, which in turns relies on the MAFFinder to locate the nearby agent system of the correct type. The implementation of the MAFAgentSystem must achieve this functionality using the MAFFinder.

Syntax

```
Location find_nearby_system_of_profile (in AgentProfile profile) raises
    (EntryNotFound);
```

Parameters

profile The profile of the agent being sent.

Exceptions

EntryNotFound The specified agent is not in the agent list for the current agent system.

Usage Notes

Sometimes an agent wishes to communicate with an object that resides in an agent system of a different type (in other words, one that does not support the agent profile of the traveling agent), or in a non-agent system. This method lets the requesting agent look for an agent system of the correct type that is closer to the object with which it wants to communicate, which resides on an incorrect type (not supporting MAF AgentSystem, not supporting the same version required, different languages, etc.). In order to be able to optimize communication with that object, another MAFAgentSystem of the correct type (the same version as required by the mobile agent) is located.

This interface is highly application specific. It would be extremely difficult to generalize on the metric for closeness. Therefore, the application must define and take advantage of this interface.

3.5.4 get_agent_status()

The get_agent_status() method returns the status of the specified agent.

Syntax

Agent_Status get_agent_status (in Name agent_name) raises (AgentNotFound);

Parameters

agent_name	Name of the agent whose status the caller wants to know.

Exceptions

AgentNotFound	The agent system could not find the specified agent.

Usage Notes

The return parameter Agent_Status can have one of three values:

- running, which means the agent is currently executing
- suspended, which means the agent is not currently executing
- terminated, which means the agent finished executing

Refer to the definition for ClassName ("MAF IDL Interfaces" on page 86) for the enumeration values of Agent_Status.

This method is useful in management applications. It allows the system manager to monitor the status of an agent.

3.5.5 get_agent_system_info()

The get_agent_system_info() method returns the AgentSystemInfo structure. This structure contains information about the agent system, including its name and the agent profile it supports.

Syntax

AgentSystemInfo get_agent_system_info();

Parameters

There are no parameters for this method. It returns information about the current agent system.

Exceptions

None.

Usage Notes

An agent can use this method to find out information about an agent system to which it wants to travel.

The AgentSystemInfo structure this method returns provides the following information about the agent system:

system_name Name of the agent system.

system_type Identifies the agent system type (for example, Aglets, MOA, or AgentTcl).

language_maps The programming language the agent system supports (for example, Java, Tcl, Scheme, or Perl), and the serialization schemes that each of these languages uses (for example, JavaObjectSerialization, ASN1_BER, ASN1_DER).

system_description Short description of the agent system. The information in this parameter is not standardized; it is implementation dependent.

major_version Version information about the agent system implementation.

minor_version Version information about the agent system implementation.get_authinfo()

serializations Identifies the serialization schemes the agent system uses (for example, JavaObjectSerialization, ASN1_BER, ASN1_DER).

3.5.6 get_authinfo()

The get_authinfo() method returns information about whether an agent was authenticated, and what authentication method was used.

Syntax

```
AuthInfo get_authinfo(in Name agent_name) raises (AgentNotFound);
```

Parameters

agent-name Name of the agent whose authentication information is requested.

Exceptions

AgentNotFound The agent system could not find the specified agent.

Usage Notes

If security is desired, the client should authenticate the agent system before calling this method.

3.5.7 get_MAFFinder()

Returns a reference to a MAFFinder for locating agents, places, and agent systems.

Syntax

```
MAFFinder get_MAFFinder() raises (MAFFinderNotFound);
```

Parameters

None.

Exceptions

FinderNotFound	Could not find the MAFFinder of the current region.

Usage Notes

Once you get the MAFFinder reference, you can use the MAFFinder methods to find agents, places, and agent systems within the region.

3.5.8 list_all_agents()

The list_all_agents() method lists all agents registered within the agent system.

Syntax

```
NameList list_all_agents();
```

Parameters

None.

Exceptions

None.

Usage Notes

This is a management operation that allows a system manager to track the agents within an agent system.

3.5.9 list_all_agents_of_authority()

The list_all_agents_of_authority() method lists all agents within the agent system that have the specified principal (authority).

Syntax

```
NameList list_all_agents_of_authority (in Authority authority);
```

Parameters

authority	Identifies the authority whose agents you want to list.

Exceptions

None.

Usage Notes

This is a management operation that allows a system manager to track the agents of a specific authority within an agent system.

3.5.10 list_all_places()

The list_all_places() method lists all places within the agent system.

Syntax

Locations list_all_places();

Parameters

None.

Exceptions

None.

Usage Notes

This is a management operation that allows a system manager to get the list of places registered with the MAFFinder.

3.5.11 receive_agent()

An agent system uses receive_agent() to receive and instantiate an agent.

Syntax

```
void receive_agent(in Name agent_name, in AgentProfile agent_profile,
      in OctetString agent, in string place_name, in ClassNameList
      class_names, in string code_base, in MAFAgentSystem agent_sender)
      raises (ClassUnknown, DeserializationFailed, MAFExtendedException);
```

Parameters

agent_name

Unique identifier for an agent. This identifier should include the authority of the person or organization the agent is representing, and the agent identity to make the agent name unique.

agent_profile

Contains information about the agent, such as the agent system type that created it, and the method used to serialize it for transfer.

agent

This parameter is opaque, and can contain anything that the sender needs to convey to the remote agent system that is not covered in the other parameters to this method. This parameter contains information that is unique to agents of a particular agent profile. The agent system receiving the agent must be able to decode the meaning of the information in this parameter.

The kinds of information that can be in this parameter include, but are not restricted to the agent class definition, the agent's execution state, and definitions of some or all the classes needed to instantiate the agent at the remote agent system.

Note that the class definitions included in this parameter affect the parameter class_names. If a class is included in the definition, it is not necessary to add it to the class_names list. Alternatively, if a class definition is not included in this parameter, it should be in the class_names list.

place_name

Name of the place where the agent will reside. If this parameter is not specified, the agent system creates the agent in a default place within the system.

class_names

List containing the name of each class necessary to instantiate the agent. Note that the class list is optional, which means it can be empty. Because CORBA does not accept null as a passed-in argument for a parameter specified as classNameList, MAF implementors must create a special null class name with the name field set to an empty string.

The list may or may not be necessary depending on the class transfer mechanism chosen. Refer to "Class Transfer" in Section 1.3.1, "Remote Agent Creation" earlier in this specification for a discussion of the various mechanisms and their requirements.

This parameter and the parameter agent are related. If the parameter agent contains a class definition that class does not need to be in the class_names list. Alternatively, any class required to instantiate the agent should be in this list if it is not included in the parameter agent.

Classes listed here may or may not need to be transferred depending on whether the agent system caches classes. In fact, the receiving agent system must decide whether fetch_class() is needed.

code_base Reference to the code base containing the necessary class definitions. The syntax of this parameter can vary between agent system types without affecting interoperability. It is returned to the requesting client software via fetch_class(), if necessary. Therefore, only the client software needs to know the syntax of this string.

agent_sender Reference to the agent system initiating the agent transfer.

Exceptions

ClassUnknown Failed to find class definition.

DeserializationFailed The agent system could not instantiate the agent because it could not decode the agent OctetString.

MAFExtendedException This is a generic exception. Use it only if no other exception applies to the error condition.

Usage Notes

One possible algorithm for the implementation of this method is:

1. Check whether the classes required to instantiate the agent are included in the receive_agent input parameter agent, or cached on the agent system platform.
2. Call fetch_class(), if necessary, to retrieve any required classes that are not available.
3. Deserialize and instantiate the agent at the place specified in the call, or in the agent system's default place if no place is specified.

The destination agent system is free to queue the agent or route it to another agent system within its region. This is an implementation detail that the specification allows but does not mandate.

3.5.12 resume_agent()

The resume_agent() method resumes execution of the specified agent.

Syntax

```
void resume_agent(in Name agent_name) raises (AgentNotFound,
      ResumeFailed, AgentIsRunning);
```

Parameters

agent_name	Name of the agent to restart.

Exceptions

AgentNotFound	Agent system could not find the specified agent.
ResumeFailed	The agent could not resume execution.
AgentIsRunning	The agent is already running; it does not need to be resumed.

Usage Notes

This method provides the management function of restarting an agent that was suspended, with suspend_agent().

3.5.13 suspend_agent()

The suspend_agent() method suspends execution of the specified agent.

Syntax

```
void suspend_agent(in Name agent_name) raises (AgentNotFound,
      SuspendFailed, AgentIsSuspended);
```

Parameters

agent_name	Name of the agent to suspend.

Exceptions

AgentNotFound	Agent system could not find the specified agent.
SuspendFailed	Could not stop agent execution.
AgentIsSuspended	The agent is already suspended.

Usage Notes

This method provides the management function of suspending execution of an agent. Use resume_agent to restart agent execution when desired. To implement this method, you should suspend the agent's execution thread and maintain the agent states.

There are several reasons for suspending an agent's execution. For example, you can suspend an agent to give resources to a higher priority thread. You can also suspend an agent if it is suspected of a security violation.

3.5.14 terminate_agent()

The terminate_agent() method stops execution of the specified agent.

Syntax

```
void terminate_agent(in Name agent_name) raises (AgentNotFound,
    TerminateFailed);
```

Parameters

agent_name	Name of the agent to terminate.

Exceptions

AgentNotFound	Agent system could not find the specified agent.
TerminateFailed	Could not stop agent execution.

Usage Notes

This method provides the management function of permanently stopping the execution thread of an agent.

3.5.15 terminate_agent_system()

The terminate_agent_system() method stops execution of the agent system.

Syntax

```
void terminate_agent_system () raises (TerminateFailed);
```

Parameters

None.

Exceptions

TerminateFailed	Could not stop agent system execution.

Usage Notes

Depending on implementation of terminate_agent_system(), the agent system may store important information, and inform all hosted agents about the intended termination.

3.6 MAFFinder Interface

The MAFFinder interface provides methods for maintaining a dynamic name and location database of agents, places, and agent systems. The interface does not dictate what method a client uses to find an agent. Instead, it provides ways to locate agents, agent systems, and places that supports a wide range of location techniques.

There are many possible ways of locating an agent. Here are four possibilities:

- Brute force search

  Find every agent system in the region, then send an agent to travel through every agent system to find the agent.

- Logging

  Whenever an agent leaves an agent system, it leaves a mark that says where it is going. Therefore, an agent system can always follow the logs to locate that agent. There should also be a way to garbage collect the logs after the agent dies.

- Agent registration

  Every agent registers its current location in a database. This database always has the latest information available about an agent's location. Note that registering the new location of an agent does add overhead to the agent go() operation. Therefore, database operations can be a bottleneck.

- Agent advertisement

  Register all the stationary places only. An agent's location is registered only when the agent advertises itself. To find a non-advertised agent, the agent system can use a brute force search or logging.

Although the MAFFinder interface does not restrict implementations to a certain set of agent location schemes, it does assume an underlying database structure that can support registering, unregistering, and locating agents, agent systems, and places.

```
interface MAFFinder {

    void register_agent (
        in Name agent_name,
        in Location agent_location,
        in AgentProfile agent_profile,) raises (NameInvalid);
    void register_agent_system (
        in Name agent_system_name,
        in Location agent_system_location,
        in AgentSystemInfo agent_system_info) raises (NameInvalid);
    void register_place (
        in string place_name,
        in Location place_location) raises (NameInvalid);
    Locations lookup_agent (
        in Name agent_name,
        in AgentProfile agent_profile) raises (EntryNotFound);
    Locations lookup_agent_system (
        in Name agent_system_name,
        in AgentSystemInfo agent_system_info)
            raises (EntryNotFound);
    Location lookup_place (in string place_name)
            raises (EntryNotFound);
    void unregister_agent (in Name agent_name)
            raises (EntryNotFound);
    void unregister_agent_system (in Name agent_system_name)
            raises (EntryNotFound);
    void unregister_place (in string place_name)
            raises (EntryNotFound);
    };
```

3.6.1 lookup_agent()

The lookup_agent method returns the locations of the specified agents. This method can search for a specific agent by name, or it can search for a set of agents that match a specific agent profile.

Syntax

```
Locations lookup_agent (
    in Name agent_name,
    in AgentProfile agent_profile) raises (EntryNotFound);
```

Parameters

agent_name	Name of the agent the client or agent wants to find.
agent_profile	Agent profile information that can be used to specify search criteria (defined in "OMG Naming Authority Identifiers" on page 43).

Exceptions

EntryNotFound	No agent could be found which matches the specified criteria.

Usage Notes

An agent can use this method to find another agent or agents with which it wants to communicate.

This is also very application specific and relates to how naming semantics is organized. In particular, various MAF implementations can organize names of families and generations of agents in different ways. Having both agent name and agent profile, allows richer search semantics, e.g. it would be possible to search for a specific family of agents, or a generation (defined by agent name) with a certain characteristic (defined by agent profile). Therefore, it is possible that both agent name and profile are non-null. That’s why there is not a union, but rather two parameters for this method.

The invoking agent can either specify the demanded agent by name, or it can specify one or more agents by using the agent_profile parameter. Note that in the latter case not all components of agent_profile must be specified. However, the components not restricted in the search must be set to zero, for an integer type, or an empty string, for types other than integer.

This method cannot guarantee that an agent will be in the Location the method returned for any specified time interval. Only an agent can control when and where it travels.

There is also a potential source of false locations. If the agent system does not remove terminated agents from the system with unregister_agent, a call to lookup_agent for that agent returns the location of an agent that is no longer available.

3.6.2 lookup_agent_system ()

Returns the location of an agent system registered with the MAFFinder. The method can search for a specific agent system by name, or it can search for a set of agent systems that match the specified AgentSystemInfo parameter.

Syntax

```
Locations lookup_agent_system (
    in Name agent_system_name,
    in AgentSystemInfo agent_system_info)
        raises (EntryNotFound);
```

Parameters

agent_system_name	Name of the agent system to locate.
agent_system_info	Agent system information that can be used to specify search criteria (refer to "OMG Naming Authority Identifiers" on page 43 for a definition).

Exceptions

EntryNotFound	No agent could be found that matches the specified criteria.

Usage Notes

This method can be used to search for agent systems that are registered with the MAFFinder.

The invoking client can either specify the demanded agent system by name, or it can specify one or more agent systems by using the agent_system_info parameter. Note that in the latter case not all components of agent_system_info must be specified. However, the components not restricted in the search must be set to zero, for an integer type, or an empty string, for types other than integer.

3.6.3 lookup_place()

Returns the location of a place registered with the MAFFinder.

Syntax

Location lookup_place (in string place_name) raises (EntryNotFound);

Parameters

place_name	Name of the place to locate.

Exceptions

EntryNotFound	The specified place is not registered with the MAFFinder.

Usage Notes

Sometimes the client software has only the name of the place to which it wants to send an agent. The client uses this method to get the location of the specified place.

3.6.4 register_agent ()

Adds the named agent to the list of agents registered with the MAFFinder. Because a mobile agent travels, this operation may be invoked very frequently by an agent in its lifetime. If this operation is invoked with an agent_name that already exists in the MAFFinder, this operation replaces the associated information (location and profile) with the information in the most recent invocation.

Syntax

```
void register_agent (
    in Name agent_name,
    in Location agent_location,
    in AgentProfile agent_profile,) raises (NameInvalid);
```

Parameters

agent_name Name of the agent to add to the list.

agent_location Location of the agent.

agent_profile Agent profile information that can be used to specify search criteria (defined in "OMG Naming Authority Identifiers" on page 43).

Exceptions

NameInvalid The request to update the MAFFinder has failed.

Usage Notes

This method provides the registration of an agent with the MAFFinder.

3.6.5 register_agent_system ()

The register_agent_system() method adds the named agent system to the list of agent systems registered with the MAFFinder. Because an agent system is a stationary object, the MAFFinder does not allow multiple invocations of this operation with the same name. When moving an agent system, unregister it before registering it again with the new location.

Syntax

```
void register_agent_system (
        in Name agent_system_name,
        in Location agent_system_location,
        in AgentSystemInfo agent_system_info) raises (NameInvalid);
```

Parameters

agent_system_name	Name of the system to add to the list.
agent_system_location	Location of the system to add.
agent_system_info	Agent system information that can be used to specify search criteria (refer to "OMG Naming Authority Identifiers" on page 43 for a definition).

Exceptions

NameInvalid	There is already an agent system registered that has the same name.

Usage Notes

This method provides agent system registration.

3.6.6 register_place ()

The register_place() method adds the location of the named place to the list of places registered with the MAFFinder. Because a place is a stationary object, the MAFFinder does not allow multiple invocations of this operation with the same name. If a place is moved, unregister it from its initial location before registering it with the new location.

Syntax

```
void register_place (
    in string place_name,
    in Location place_location) raises (NameInvalid);
```

Parameters

place_name	Name of the place to add to the list.
place_location	Location of the place.

Exceptions

NameInvalid	There is already a place registered that has the same name.

Usage Notes

This method is one of a group of methods you can use to maintain a list of places with the MAFFinder.

3.6.7 unregister_agent ()

Removes the specified agent from the list of agents that are registered with the MAFFinder.

Syntax

```
void unregister_agent (in Name agent_name) raises (EntryNotFound);
```

Parameters

agent_name	Name of the agent to remove from the list of agents.

Exceptions

EntryNotFound	The specified agent is not registered with the MAFFinder.

Usage Notes

This method is one of a group of methods you can use to maintain a list of agents within an agent system.

3.6.8 unregister_agent_system ()

Removes the specified agent system from the list of agent systems registered with the MAFFinder.

Syntax

```
void unregister_agent_system (in Name agent_system_name) raises
      (EntryNotFound);
```

Parameters

agent_system_name	Name of the system to remove from the list.

Exceptions

EntryNotFound	The specified agent system is not registered with the MAFFinder.

Usage Notes

This method is one of a group of methods you can use to maintain a list of agent systems registered with the MAFFinder.

3.6.9 unregister_place ()

Removes the specified place from the list of places registered with the MAFFinder.

Syntax

```
void unregister_place (in string place_name) raises (EntryNotFound);
```

Parameters

place_name	Name of the place to remove from the list.

Exceptions

EntryNotFound	The specified place is not registered with the MAFFinder.

Usage Notes

This method is one of a group of methods you can use to maintain a list of places registered with the MAFFinder.

MAF Scenario 4

This section presents a real world scenario for agent deployment. It incorporates most of the interfaces described in this proposal.

4.1 Overview

FAM Corp has grown rapidly and expanded its retail business into every state, partly by building new stores and partly by acquiring smaller companies. They now have a dozen major regional centers, and each one of these has close ties with a dozen or more stores in the region. FAM offers a wide range of products and services for home and office, and has recently added software rental to its services.

Each regional center has its own inventory system, more or less coordinated with the stores in its region. However, part numbers are not fully standardized, and central IT says it will be years before the different systems are fully integrated. Adding to the problem is the continuing fast growth of the corporation.

4.2 The Problem

Quality, price, and, above all, service is the FAM corporate credo. If a customer asks for an item in a store that is not in stock, the CEO insists that the store check with all other stores to see if it is in stock or on order elsewhere. But with their continuing growth and disparate computer systems, this is difficult to do. Sometime, these checks are aggravated by problems, such as a communications link in their network that is down, heavy traffic on their network due to the daily replication and update of the databases in several regional centers, and legacy systems on multiple platforms that have inconsistent part number and vendor number assignments. New stores and new services are not being integrated quickly enough to maintain the level of service the CEO demands, and their customers expect.

4.3 The Solution Today

A customer comes into the Boston store, and asks for twenty blue widgets. Joe, the salesperson in the store, checks the local inventory on his terminal. Not in stock. He can check also other regional stores on-line, although that doesn't reflect replenishment orders recently placed, but nothing there either. He calls his contact Susan at another regional center, who looks up the inventory there (no luck), and while Joe is on the phone, she calls two other stores just acquired last month. One of them has the item, but only five of them, not the twenty the customer wants.

Joe calls three more regional centers without success, though a person at one of the regional centers says he will call back in the afternoon when his computer link is fixed. Joe knows the remaining regional centers are very unlikely to have this item, and he has other customers waiting. He asks the customer to call him tomorrow, and tells him that he should have more information for him then. If they locate the widgets in another regional center, they can be shipped to this store for pickup in about a week. The customer leaves the store and never calls back.

4.4 The Solution Tomorrow

A customer comes into the Boston store, and asks for twenty blue widgets. Joe, the salesperson in the store, checks the local inventory on his terminal. Not in stock. The terminal tells him it is starting to search other stores to see if they have blue widgets in stock or on order. Does he want to cancel the search? Does he want information about similar items, other colors, or different sizes? Joe answers that he does. He is told that he will be given partial results every two minutes, and a chance to modify the search at that time.

Two minutes later Joe sees that a store in Atlanta has five of the widgets, and a store in Charlotte has forty green ones in a smaller size. The customer says yellow or red would be okay, but green is not. Joe modifies the search criteria and waits.

The customer meanwhile has just called his spouse on his cellphone. He is sorry but red is not okay. Joe suspends the search. After a few minutes, the customer says the widgets must be blue or green. Joe modifies the search criteria again, then tells the system to restart the search from the prior point

After four minutes, the terminal displays information that a store in Dallas has fifty blue widgets. The customer asks for them to be shipped directly to his home. Joe enters the order, which will be filled by the Dallas store, and shipped UPS the next day.

4.5 Behind The Scenes ....

The IT group has decided that mobile agent technology, combined with better collection and integration of data within a store or regional group of stores, offers their company the best opportunity for handling rapid growth while maintaining, and, indeed improving, customer service. Each store has installed a MAF agent system. There are several different hardware platforms involved, but the MAF agent systems interoperate, providing the infrastructure to support mobile agents in a heterogeneous environment. Within this environment, agents can efficiently find information, and coordinate with each other and the controlling user.

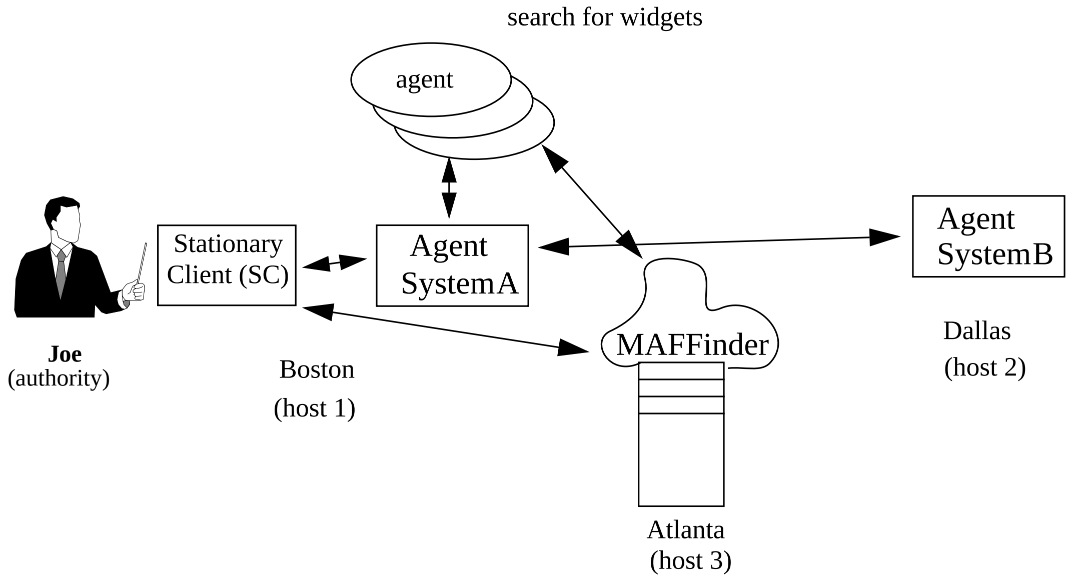


Figure 4-1 Key Actors in the Scenario.

The following entities are involved: ***agents*** *execute on top of* ***agent systems****; agents and agent systems use the* ***MAFFinder*** *to locate other agent systems; the stationary client, using interfaces for an agent system and MAFFinder, can create an agent with Joe's authority to perform specific tasks for it; the stationary client can also monitor and control the agent it created, because FAM's policy allows agents of the same authority to manage each other. The agent system and MAFFinder interfaces can be invoked by the following 'clients': a mobile agent, or a stationary client (user application) that monitors/controls agents, or other agent systems.*

The MAF services this scenario uses are:

- locating an agent system for agent creation (using the MAFFinder interface).
- creating an agent to begin the query for twenty widgets (using the MAFAgentSystem interface).
- the agent registers itself as it moves (using the MAFFinder interface), so that others can find it.
- the agent goes from place to place to look for the requested widgets (using the MAFAgentSystem interface).
- finding the search agent to request status (using the MAFFinder interface).
- suspending or resuming the agent (using the MAFAgentSystem interface).

The main actors in the scenario are described in Figure 4-1. Joe our salesperson is the authority. The user application interacting with Joe at his terminal is the stationary client (SC). The SC will control/monitor all agent activity in this scenario.

4.6 Overview of interaction with MAF

Joe's request starts a series of actions. This section provides an overview of the entire interaction. The indented text describes the underlying actions of the SC, and lists the MAF services that are invoked. Further details are presented in double-indented text.

The overview shows how Joe and his customer get the information they need in a timely manner, when the user application (SC) working on their behalf invokes the MAF services to dispatch and control mobile agents going to the appropriate sites.

Please note that in all cases where a MAFFinder is used to locate a stationary object such as stationary agents, agent systems, and places, the client actually can choose between the CORBA naming service and the MAFFinder. For simplicity, this section only mentions MAFFinder. In the case where a MAFFinder should be located, one can get the object reference of the MAFFinder via MAFAgentSystem::get_MAFFinder(), or via the CORBA naming service.

Joe is informed that the item is not in stock locally, and a search has started....

> [Locate]. The SC begins by obtaining a reference to a 'finder' (MAFFinder) by invoking a get_MAFFinder() method on behalf of an agent system (MAFAgentSystem). This method takes no parameters. A finder can be shared among multiple agent systems. An agent system and MAFFinder can be implemented in a single module or in two separate modules. It is left open to the reference implementation how/whether to combine them.

The SC obtains the locations of available agent systems and places using the MAFFinder methods lookup_agent_system() and lookup_place(). This information is used to decide where to start an agent, and where to move it during its lifetime. To locate other agent systems and places, the SC needs to know only their names, which are provided as parameters to the methods lookup_agent_system() and lookup_place(). Location transparency is achieved this way. The SC can also obtain additional information about each agent system by invoking the get_agent_system_info() method (MAFAgentSystem interface). This information is agent-system specific. This method takes no parameters, and it returns agent-system-specific information. The lookup operation of an agent system is based either on the agent system name or the agent system information.

When a new agent system starts up, it registers itself with the MAFFinder using the register_agent_system() method. This method takes the agent system name, agent system information, and the location as parameters. The agent system information parameter (agent_system_info) is used during lookup to identify agent systems with specific characteristics. It is an internal matter of the MAFFinder to announce itself available to other MAFFinders, as well as to announce availability of the new agent system. Similarly, if the agent system is terminated, or if its services cease to be available for clients, the agent system can be unregistered at the MAFFinder using unregister_agent_system(), which requires only the agent system name as a parameter. If a client has a reference only to an agent system, it can obtain the reference to MAFFinder by invoking the get_MAFFinder() method on behalf of the agent system. An agent system can be located using its name, by invoking the lookup_agent_system() method on the MAFFinder. Similarly, multiple agent systems can be found by searching based on agent system information.

[Create]. Based on the availability of other agent systems and places, the SC creates an agent on a selected agent system by invoking the create_agent() method of the MAFAgentSystem interface.

As a part of this invocation, the SC provides various parameters. The agent_profile parameter specifies the agent and the originating agent system specific information, such as the type of the agent system, and the way the agent was serialized. The agent parameter contains agent state, such as the serialized state of an agent's objects, and agent class definitions. The receiving agent system uses this data to create a new instance of the agent. If specified, the place_name is the destination of the agent on the receiving host. Otherwise, the agent is associated with the agent system's default place. The Arguments parameter is passed to the agent constructor. class_names, code_base and class_provider are used to retrieve agent classes. The create_agent() method returns the name of the agent created.

[Register]. Once a new agent is created, the SC (as well as the agent), has many interfaces available to control the activity and movement of the agent throughout its life. The agent can also control the activity of other agents.

To allow other SCs and other agents to be aware of it and locate it, the agent needs to register itself with the MAFFinder (if the controlling SC wants registration). This is achieved using the register_agent() method of the MAFFinder interface, which takes as parameters the agent name, agent profile and the agent's current location. Agent profile specifies agent characteristics that can be useful for filtering during a lookup operation. Similarly, if the agent does not want to publicize itself anymore, it can remove itself from the MAFFinder, by invoking unregister_agent(). This method takes only the agent name as a parameter. Similar methods can be invoked for registering and looking up a place at a MAFFinder, using methods register_place() and unregister_place() respectively. The register_place() method takes the place name as a parameter, and the unregister_place() method takes the location.

The SC acting on behalf of Joe wants the agent to go to each regional center, and get information about relevant stock items. Once at a regional center, the agent may decide (based on the information found at that site) to visit other stores in that region before going to the next regional site. When the SC creates the agent, it gives the agent an initial list of sites to visit (its itinerary). Either the SC or the agent can modify this list later, based on information it gathers or receives from an agent system.

[Move]. When the agent completes its work at the first site (host), it migrates to the next host on its list. Note that it is the agent itself that initiates migration, not the SC or agent system. While it is possible to achieve an agreement between an agent and other clients, whereby they can initiate migration in some cases, the MAF specification makes neither provisions nor guarantees for this.

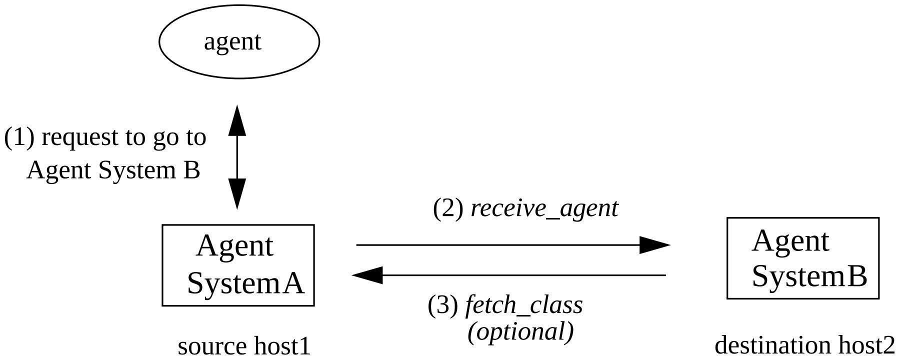


Figure 4-2 This shows the sequence of events during agent's migration from host1 to host2. (1) The agent requests from the Agent System A on the source host to migrate to destination host, to Agent System B (2). The Agent System A invokes a receive_agent method on the Agent System B, passing various parameters relevant for migration. (3) When agent successfully migrates, the Agent System B downloads leftover classes from the source host, by invoking fetch_class method on Agent System A.

An agent initiates its migration by contacting its current agent system. The current agent system then invokes the receive_agent() method on the agent system where the agent wants to go.

The agent_name is passed as a parameter. This is the name obtained when the agent was created. The parameters agent, place_name, agent_profile, class_names and code_base have the same meaning as in create_agent(). Finally, the agent_sender represents the agent system the agent is leaving (source). When attempting to receive an agent, the destination needs to fetch from the source agent system the classes the agent requires that do not exist at the destination system. This is achieved using the method fetch_class, which uses as parameters class_names and code_base. Previously, these parameters were passed to the destination host as parameters of the receive_agent() method. The destination agent system must either fetch all the classes the agent's code requires immediately, or fetch these classes as the agent's code requires them. Figure 4-2 describes the scenario of migrating an agent from the source to destination host.

After a set period of time (about two minutes), the SC wants to retrieve the information the agent has gathered so far, then pass it on to Joe.

[Status]. The agent has visited several sites so far, and gathered relevant information from those sites about the particular stock items. Some of these stock searches done locally by the agent are in fact quite complex, processing a lot of data from several databases on interconnected systems, and making a summary of the requested information for later transmission to the SC.

The SC must find and interrogate the agent for its status, and also obtain any relevant data that the agent wants to send back to the SC through the non-MAF interfaces. The SC obtains the status by invoking get_agent_status(), and using

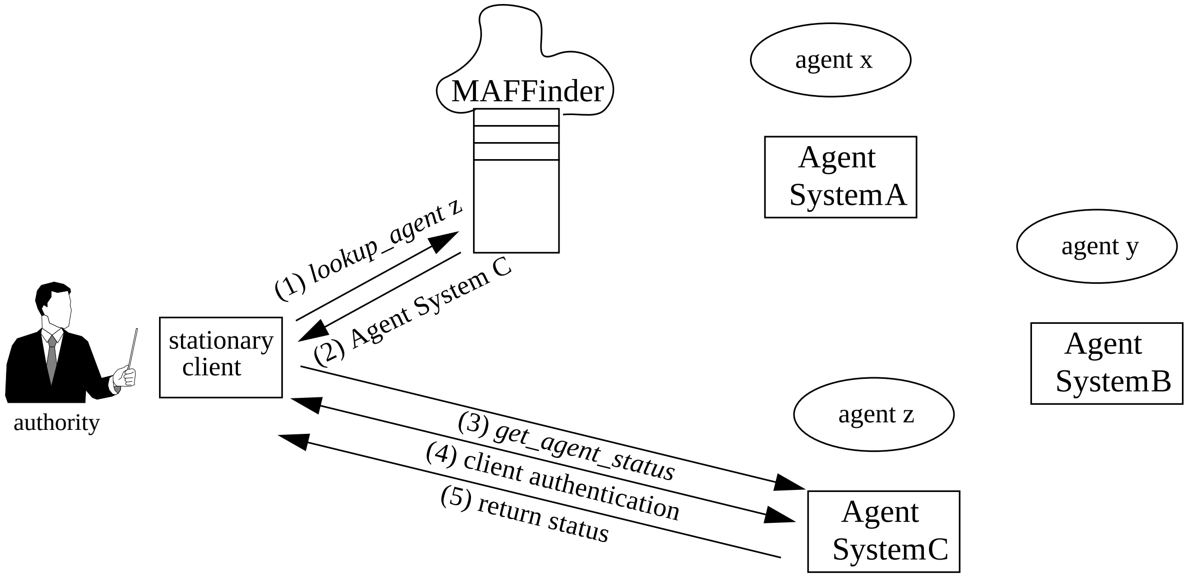


Figure 4-3 In order to monitor an agent, its authority first must locate an agent by contacting the MAFFinder. Once the agent's location is found, the authority contacts the agent system where the agent currently resides. Before providing any information about the agent, it might be required to authenticate the requesting client first. Only then is the information about the agent provided.

as a parameter the name of the agent in question. It is also possible to find out if the agent was authenticated, and which authentication method was used. This is achieved by invoking the get_authinfo() method.

The SC can also find out all the places and agents residing at an agent system by invoking the methods list_all_agents() and list_all_places(), respectively. These two methods return the lists of the agent and place names. It is also possible to obtain all the agents that belong to a user (distinguished by its principal) by invoking the method list_all_agents_of_authority(), passing the authority as a parameter.

In some cases, however, it is required that the invoking client be authenticated prior to being allowed to invoke this method. Prior to being able to invoke these methods, the client needs to resolve the agent's current location, by invoking the lookup_agent() method (MAFFinder). The agent must maintain this information by updating appropriate MAFFinders. Lookup can be based either on agent name or on agent profile. Figure 4-3 describes the steps involved in monitoring an agent by its authority.

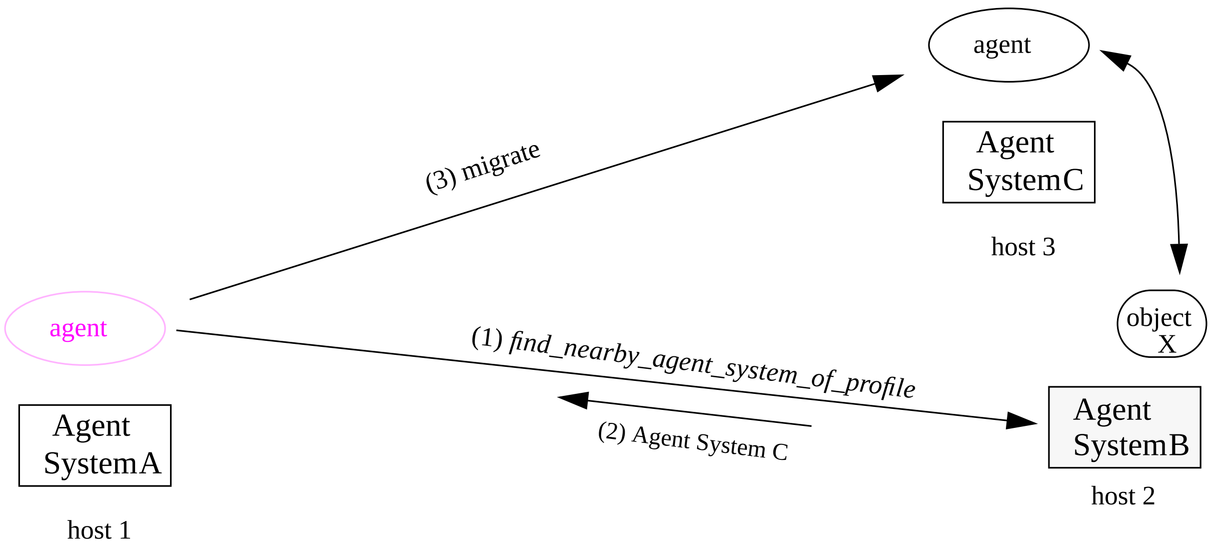


Figure 4-4 If the agent cannot migrate for compatibility reasons to a host (host 2) where an object of interest resides, then the agent can obtain information about the closest compatible system, migrate to that system (host 3), then benefit from the locality of reference in accessing object X.

Joe sees new information on his terminal after two minutes. A store in Atlanta has five of the widgets, and one in Charlotte has forty green ones in a smaller size. The customer says yellow or red would be okay, but green is not. Joe modifies the search criteria and waits.

The SC sees from the data that the agent was not able to visit the Miami site. It was on the itinerary, but the agent system in Miami was not able to receive agents at that time. The SC therefore locates another store in the Miami region that has a compatible agent system, and instructs the agent to go there.

In order to access an object on an agent system that cannot receive the agent, the agent can invoke a method find_nearby_agent_system_of_profile() on the incompatible agent system. This method takes as a parameter the profile describing the agent system. It returns the location of the closest agent system of compatible type, if this information/system is available. This type of information is useful for improving the locality of reference between the object and the agent in question. Using the find_nearby_agent_system_of_profile() method is described in Figure 4-4.

To relay the modified search criteria (yellow or red is okay, but green is not), the SC in fact passes the new data to the agent directly, using the agent application interface, which MAF does not cover.

The customer meanwhile has just called his spouse on his cellphone. He is sorry but red is not okay. Joe suspends the search. After a few minutes, the customer says it must be blue or green. Joe modifies the search criteria again, then tells the system to restart the search from the prior point.

> Using the agent name, the SC suspends the agent by invoking the method suspend_agent(). Note that clients other than the SC (the owner), need to be authenticated and authorized to be able to invoke the suspend() and resume() methods.
>
> Some time later, the SC tells the agent to go back to the prior site (where the search with the 'yellow or red is okay' criteria started), and resume the search with the modified search criteria. This is done by invoking application specific methods followed by the resume_agent() method on behalf of the agent. Then, the agent invokes a receive_agent() method to move back to a specified checkpoint stored by the agent, and continues the search using the new criteria.

After four minutes the terminal displays information that a store in Dallas has fifty blue widgets. The customer asks for them to be shipped directly to his home. Joe enters the order, which will be filled by the Dallas store and shipped UPS the next day.

> The order transaction is sent directly from the SC to a static agent which deals with orders at the Dallas site. The mobile agent has fulfilled its duty, so the SC (or other authorized client) can now terminate it, by invoking the terminate_agent() method. This is the end of the agent life cycle.

References 5

1. OMG, The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995.

2. OMG, Common Facilities RFP 3, OMG TC Document 95-11-3, November 3, 1995.

3. Aglets Workbench (http://www.trl.ibm.co.jp/aglets).

4. Mubot: (http://www.crystaliz.com).

5. Odyssey: (http://www.genmagic.com/agents/).

6. Mobile Objects and Agents (http://www.opengroup.org/RI/java/moa/index.htm).

MAF IDL Interfaces A

```
module CfMAF {

    /**************************************************************************************/
    /*              Data Types                                            */
    /**************************************************************************************/

    typedef sequence<octet>              OctetString;
    typedef sequence<OctetString>        OctetStrings;

    typedef OctetString     Authority;
    typedef OctetString     Identity;

    typedef short                             LanguageID;
    typedef short                             AgentSystemType;
    typedef short                             Authenticator;
    typedef short                             SerializationID;
    typedef sequence<SerializationID>         SerializationIDList;

    typedef any Property;
    typedef sequence<Property>                PropertyList;

    struct Name {
        Authority               authority;
        Identity                identity;
        AgentSystemType         agent_system_type;
    };
    typedef sequence<Name>       NameList;
```

```
struct AuthInfo {
      boolean is_authenticated;
      Authenticator authenticator;
};

struct LanguageMap {
      LanguageID                     language_id;
      SerializationIDList            serializations;
};
typedef sequence<LanguagedMap>             LanguageMapList;
struct AgentSystemInfo {
      Name                           agent_system_name;
      AgentSystemType                agent_system_type;
      LanguageMapList                language_maps;
      string                         agent_system_description;
      short                          major_version;
      short                          minor_version;
      PropertyList                   properties;
};
struct AgentProfile{
      LanguageID                     language_id;
      AgentSystemType                agent_system_type;
      string                         agent_system_description;
      short                          major_version;
      short                          minor_version;
      SerializationID                serialization;
      PropertyList                   properties;
};

struct ClassName{
      string              name;
      OctetString         discriminator;
};
typedef sequence<ClassName> ClassNameList;
typedef sequence<octet>                      Arguments;
typedef string                               Location;
typedef sequence<Location>                   Locations;
enum AgentStatus {
      CfMAFRunning, CfMAFSuspended, CfMAFTerminated
};
```

```
/********************************************************************/
/*          Exceptions                                              */
/********************************************************************/

exception AgentNotFound {};
exception AgentIsRunning {};
exception AgentIsSuspended {};
exception ArgumentInvalid {};
exception ClassUnknown {};
exception DeserializationFailed {};
exception EntryNotFound {};
exception FinderNotFound {};
exception MAFExtendedException {};
exception NameInvalid {};
exception ResumeFailed {};
exception SuspendFailed {};
exception TerminateFailed {};


/********************************************************************/
/*          Interface Definitions          */
/********************************************************************/
interface MAFFinder {

    void register_agent (
        in Name agent_name,
        in Location agent_location,
        in AgentProfile agent_profile,) raises (NameInvalid);
    void register_agent_system (
        in Name agent_system_name,
        in Location agent_system_location,
        in AgentSystemInfo agent_system_info) raises (NameInvalid);
    void register_place (
        in string place_name,
        in Location place_location) raises (NameInvalid);
    Locations lookup_agent (
        in Name agent_name,
        in AgentProfile agent_profile) raises (EntryNotFound);
    Locations lookup_agent_system (
        in Name agent_system_name,
        in AgentSystemInfo agent_system_info)
            raises (EntryNotFound);
    Locations lookup_place (in string place_name)
            raises (EntryNotFound);
    void unregister_agent (in Name agent_name)
            raises (EntryNotFound);
    void unregister_agent_system (in Name agent_system_name)
            raises (EntryNotFound);
    void unregister_place (in string place_name)
            raises (EntryNotFound);
};
```

```
    interface MAFAgentSystem {

        Name create_agent (in Name agent_name,
            in AgentProfile      agent_profile,
            in OctetString       agent,
            in string            place_name,
            in Arguments         arguments,
            in ClassNameList     class_names,
            in string            code_base,
            in MAFAgentSystemclass_provider) raises (ClassUnknown,
                ArgumentInvalid, DeserializationFailed,
                MAFExtendedException);

        OctetStrings fetch_class(in ClassNameList class_name_list,
            in string code_base, in AgentProfile agent_profile) raises
            (ClassUnknown, MAFExtendedException);
        Location find_nearby_agent_system_of_profile (in AgentProfile
            profile) raises (EntryNotFound);
        AgentStatus get_agent_status(in Name agent_name)
            raises(AgentNotFound);
        AgentSystemInfo get_agent_system_info();
        AuthInfo get_authinfo(in Name agent_name) raises (AgentNotFound);

        MAFFinder get_MAFFinder() raises (FinderNotFound);
        NameList list_all_agents();
        NameList list_all_agents_of_authority(in Authority authority);
        Locations list_all_places();

        void receive_agent(in Name agent_name,
            in AgentProfile      agent_profile,
            in OctetString       agent,
            in string            place_name,
            in ClassNameList     class_names,
            in string            code_base,
            in MAFAgentSystemagent_sender) raises (ClassUnknown,
                ArgumentInvalid, DeserializationFailed,
                MAFExtendedException);

        void resume_agent(in Name agent_name) raises (AgentNotFound,
            ResumeFailed, AgentIsRunning);
        void suspend_agent(in Name agent_name) raises (AgentNotFound,
            SuspendFailed, AgentIsSuspended);
        void terminate_agent(in Name agent_name) raises (AgentNotFound,
            TerminateFailed);
    };
};
```

Assigned Numbers B

This appendix recommends to the OMG naming authority numbers for the currently available agent system types, languages, encoding mechanisms, and authentication methods.

B.1 Languages

The assigned numbers for languages are:

- LanguageNotSpecified (0)
- Java (1)
- Tcl (2)
- Scheme (3)
- Perl (4)

B.2 Agent System Types

The assigned numbers for agent system types are:

- NonAgentSystem (0)
- Aglets (1)
- MOA (2)
- AgentTcl (3)

B.3 Encoding Mechanisms

The assigned numbers for authenticator types are:

- none (1)
- one-hop (2)

B.4 Serialization Methods

The assigned numbers for encoding mechanisms are:

- SerializationNotSpecified (0)
- Java Object Serialization (1)