

CERN/IT/ASD/RD45/98/12
Javier Conde
16 December 1998



Mobile Agents in JAVA



1 INTRODUCTION.....	6
2 WHY MOBILE AGENTS.....	6
3 JAVA AS TECHNOLOGY BASE FOR MOBILE AGENTS.....	7
4 IBM'S AGLETS.....	8
5 MITSUBISHI'S CONCORDIA.....	12
6 GENERAL MAGIC'S ODYSSEY.....	18
7 OBJECTSPACE'S VOYAGER.....	20
8 OTHER AGENT SYSTEMS IN JAVA.....	27
8.1 JAVA AGENT TEMPLATE.....	27
8.2 NCSA HABANERO.....	28
9 OTHER TECHNIQUES FOR DISTRIBUTED COMPUTING.....	28
9.1 THE COMMON OBJECT REQUEST BROKER (CORBA).....	29
9.2 RMI.....	34
9.3 SERVLETS.....	36
10 AGENT SYSTEMS COMPARISON.....	38
11 SOME APPLICATIONS OF MOBILE AGENT TECHNOLOGY IN HEP ENVIRONMENTS.....	42

1 Introduction

The aim of this paper is to explain how to implement a distributed environment using Java agents and to provide a comparison between different implementations of agent based systems and other distributed technologies in Java.

The purpose of this study was to choose one implementation for a distributed agent model, for controlling and managing the servers involved in an Objectivity/DB federated database, and for accessing the information stored in the federated database.

First of all, we will discuss the advantages of mobile agents and we will give an overview of some different implementations of Java agents, and comment the main features of each one. There are also some other techniques to develop distributed programming that we will discuss.

The aim of this paper is not to study of all the possibilities of the use of agents, but only to see the improvements that we will obtain in our applications using agents.

We then discuss different applications using mobile agent systems and Objectivity/DB Java bindings, for accessing the information locally, instead of using a WAN communication.

For a further discussion of the use of agents see “Computational Media for Mobile Agents”, Nelson Minar, available at <http://nelson.www.media.mit.edu/people/nelson/research/dc/dc.html>.

2 Why mobile agents

The traditional paradigm that ties all distributed object technologies together is a synchronous message-passing paradigm whereby all objects are distributed, but stationary, and interact with each other through message-passing.

Mobile agents can provide a single uniform paradigm for distributed object computing, encompassing synchrony and asynchrony, message-passing and object-passing, and stationary objects and mobile objects.

Along with mobility, agents have the following unique and important computational characteristics:

- **Object-passing:** when a mobile agent moves, the whole object is passed; that is, its code, data, execution state, and travel itinerary are passed together.

- **Autonomous:** the mobile agent contains sufficient information to decide what to do, where to go, and when to go.
- **Asynchronous:** the mobile agent has its own thread of execution and can execute asynchronously.
- **Local interaction:** The mobile agent interacts with other mobile agents or stationary objects locally.
- **Disconnected operation:** the mobile agent can perform its tasks whether the network connection is open or closed. If the network connection is closed and it needs to move, it can wait until the connection is reopened.
- **Parallel execution:** more than one mobile agent can be dispatched to different sites to perform tasks in parallel.

The technical advantages of mobile agents are many, and there is no single alternative to all of the functionality they provide.

3 Java as Technology Base for Mobile Agents

There are many technical challenges to implementing mobile agent systems. Most of these problems are in the structure of the computational medium, the environment the agents operate in. Servers must be designed, implemented, and deployed that not only allow mobile agents to run, but allow them to run safely.

The main requirements for mobile agent systems are:

- **Portability:** mobile agent code must be portable; when an agent arrives at a server the server needs to be able to execute that agent. Commonly used computer languages such as C and C++ are not very portable.
- **Network communication:** Mobile agents that live in the network need to be written in a language that makes network access simple. It must be easy to transfer objects across the network and to invoke methods of remote objects.
- **Server security:** a major concern specific to mobile agents is the protection of the servers running the agents. Running arbitrary programs on a machine is dangerous. Server execution environment can be designed to make dangerous operations difficult or impossible. An approach involves creating a "sandbox" for visiting agents, restricting access to resources and ensuring the agent cannot escape those restrictions.

No system perfectly meets all the requirements, but the Java system from Sun Microsystems is the best available system today. The features included in Java, and needed by mobile agent systems are:

- **Portability:** Java takes a virtual machine approach to portability. Java programs are shipped as bytecode; the bytecodes are a simple stack language that is interpreted by a virtual machine on the compute server.

- **Network communication:** One of Java's main strengths is that it gives simple access to Internet communication. Source code for software can be transparently downloaded from anywhere on the Internet. Custom socket level transmission of byte streams is easy with existing Java class libraries. Java 1.1 includes two standard facilities to make distributed object easy: object serialisation and remote method invocation (RMI).
- **Server security:** A special object, a security manager, defines which resources a Java program is allowed to access. The Java virtual machine also contains a bytecode verifier that does static checks to prevent forbidden code sequences from being loaded, thereby ensuring the unbreachability of the sandbox surrounding the incoming code.

There are some other technologies that also support mobile agents. One of the first agent systems is Telescript from General Magic. Other implementations are Inferno (a network operating system based on Plan 9, from Lucent Technologies) and ActiveX from Microsoft.

4 IBM's Aglets

The Aglet Workbench is a 100% pure Java mobile agent technology. This package has been developed at the IBM Tokyo Research Laboratory (<http://www.trl.ibm.co.jp/aglets/>).

Aglets are Java objects that can move from one host to another. The Java Aglet API (J-AAPI) is the interface to build aglets and their environment. J-AAPI defines the methods for aglet creation, message handling, dispatching, retraction, deactivation/activation, cloning, and disposing of the aglet. This API is platform-independent and needs the JDK 1.1 or later to run.

The documentation that comes with the Aglet includes:

- **Installation guide:** this guide will take you through the initial steps of installing and setting up the environment.
- **Getting started:** general introduction to the ASDK (Aglets Software Development Kit), and how to start Tahiti (the Aglets server).
- **Release Notes:** features of this release
- Demonstration programs and samples.

There are also three other papers in the Aglets web page: [Aglets Specification](#), [Agent Transfer Protocol](#) (an application level protocol for distributed agent-based systems), and the [FAQ](#).

As all the agent implementations, the Aglets system has an agent server called Tahiti. By default, it starts using the port 434 for communications, but it can be modified. For communicating between the servers, it uses the Agent Transfer Protocol (ATP) that offers a platform independent protocol for transferring agents between networked computers.

The purpose of ATP is to offer a simple and platform independent protocol for transferring agents between networked computers. ATP also offers the opportunity to handle agent mobility in a general and uniform way, regardless of the agent implementation language and vendor specific platform.

ATP defines the following four standard request methods:

- **Dispatch:** the dispatch method requests a destination agent system to reconstruct an agent from the content of a request and to start executing the agent. If the request is successful, the sender must terminate the agent and release any resources consumed by it.
- **Retract:** the retract method requests a destination agent system to send a specified agent back to the sender. The receiver is responsible for reconstructing and resuming the agent. If the agent is successfully transferred, the receiver must terminate the agent and release any resources consumed by it.
- **Fetch:** the fetch method is similar to the GET method in HTTP; it requests a receiver to retrieve and send any identified information (normally class files).
- **Message:** the message method is used to pass a message to an agent identified by an agent-id and to return a reply value in the response. Although the protocol adopts a request/reply form, it does not lay down any rules for a scheme of communication between agents.

Tahiti uses a graphical user interface to monitor and control aglets executing on the server. You can start new aglets, know the aglets that are running on your system at any time, deploy, retract and kill aglets. Additionally, the user can ask the server to display information on memory usage, thread state and log messages.

The `Aglet` abstract class defines the fundamental methods for a mobile agent to control their mobility and lifecycle. Only classes extending the `Aglet` class can be moved across the network. When a new aglet is created some attributes with information about the creator and the host where it is created are stored in an `AgletInfo` variable. Once you create an `Aglet` class, you can send it to any other host. Agent creation is always local.

When an aglet wants to communicate with other aglets, it has to first obtain the proxy to the object. This proxy is an interface that acts as a handle of an aglet and provides a common way of accessing the aglet behind it. Public aglet methods cannot be accessed directly from other aglets for security reasons. When the `AgletProxy` is invoked, it consults the Security Manager to determine whether the current execution context is permitted to perform the method.

The context of an aglet is a proxy to the runtime environment that it occupies. This context is used to obtain local information such as addresses of the hosting context, proxies to the aglets in the same context, and for create a new aglet in the context.

Aglet objects communicate by exchanging objects of the `Message` class. A message object has a `String` object to specify the kind of the message and an arbitrary number of arguments. An aglet that wants to talk to another aglet first has to create a message object, then send it to the peer aglet. The receiver aglet has to define its `handleMessage` method to

handle the incoming messages. In the `handleMessage` method, a `Message` object is passed as argument and can be used to perform the operation according to the kind of message. If it is handle, the method has to return a boolean `true` value. If the message cannot be handle, this method has to return `false`, and the sender receives a `NotHandleException`.

There are three different types of messages:

- **Now-type:** a now-type message is synchronous and blocks until the receiver has completed the handling of the message.
- **Future-type:** a future-type message is asynchronous and does not block the current execution. The method returns a `FutureReply` object that can be used to obtain the result or wait for it later.
- **Oneway-type:** an oneway-type message is asynchronous and does not block the current execution. It differs from a future-type message in the way it is placed at the tail of the queue even if it is send to the aglet itself, and does not have return value.

If an aglet send a message to itself, the message is not put at the tail of the queue but at the head, and it is executed immediately to avoid deadlock.

Behaviour supported in the aglet object model includes *creation, cloning, dispatching, retraction, deactivation, activation, disposal of* and *messaging*.

- The **creation** of an aglet takes places place in a context. The new aglet is assigned an identifier, inserted in the context, and initialised. The aglet starts executing as soon as it has successfully been initialised.
- The **cloning** of an aglet produces an almost identical copy of the original aglet in the same context. The only differences are the assigned identifier and that execution restarts in the new aglet. Execution threads are not cloned.
- **Dispatching** of an aglet from one context to another will remove it from its current context and insert it into the destination context, where it will restart execution (execution threads will not migrate).
- The **retraction** of an aglet will remove it from its current context and insert it into the context from which the retraction was requested.
- The **deactivation** of an aglet is the ability to temporarily remove it from its current context and store it in secondary storage. Activation of an aglet will restore it in a context.
- The **disposal** of an aglet will halt its current execution and remove it from its current context.
- **Messaging** between aglets involves sending, receiving and handling messages synchronously as well as asynchronously.

To specify the aglet behaviour in response of the different events, you can overwrite the next methods. All these callbacks are also inserted into the message queue. You will not receive the event until the current message owning the monitor is completed.

When	Event Object	Listener	Method called
About to be cloned	CloneEvent	CloneListener	onCloning
Clone is created	CloneEvent	CloneListener	onClone
After the clone was created	CloneEvent	CloneListener	onCloned
About to be dispatched	MobilityEvent	MobilityListener	onDispatching
About to be retracted	MobilityEvent	MobilityListener	onReverting
After arrived at the destination	MobilityEvent	MobilityListener	onArrival
About to be deactivated	PersistencyEvent	PersistencyListener	onDeactivating
After activated	PersistencyEvent	PersistencyListener	onActivation

Security is essential to any mobile agent system, because accepting a hostile agent may lead to your computer being damaged or your privacy intruded upon. For secure agent execution, the agent system must provide the following security services:

- Authentication of the Sender, the Manufacturer and the Owner of the Agent.
- Authorization of the Agent (or Its Owner)
- Secure Communication between Agent Systems.
- Non-repudiation and Auditing.

The security architecture implements the security model by providing a set of components and their interfaces. Any useful mobile agent system must implement general and flexible security policies. The Aglet Workbench security model simplifies the administration of these policies by introducing the notion of roles, namely, the manufacturer, owner and master. There is a language for defining policies that provides named groups, composite principals (a set of principal or entity whose identity can be authenticated by a system) and hierarchical resources with associated permissions that allow the definition of high-level authorisation policies. For more information, see article about the “Security Model for Aglets” in IEEE July-august 1997.

Although it should be possible to directly edit the policy file to specify permissions, Tahiti provides you with the GUI that make easy to define the security group and set permissions.

The Aglets Workbench does provide a reasonable level of security to make it safe to use mobile agent applications. The following security features are supported in the latest Aglets runtime:

- Authentication of users and domains.
- Integrity checked communication between servers within a domain.
- Fine-grained authorization similar to the JDK1.2 security model.

The Aglets runtime itself has no communication mechanism for transferring the serialized data of an aglet to destinations. Instead, the Aglets runtime uses the communication API that abstracts the communication between agent systems. This API defines methods for creating and transferring agents, tracking agents, and managing agents in an agent-system- and protocol-independent way.

The current Aglets uses the Agent Transfer Protocol (ATP) as the default implementation of the communication layer. ATP is modeled on the HTTP protocol, and is an application-

level protocol for transmission of mobile agents. To enable remote communication between agents, ATP also supports message passing.

The communication API used by Aglets runtime is derived from the OMG standard, MASIF (Mobile Agent System Interoperability Facility), which allows various agent systems to interoperate. This interface abstracts the communication layer by defining interfaces and providing a common representation in Java that conforms to the IDL defined in the MASIF standard.

Although MASIF interfaces are intended for CORBA objects, the interfaces actually defined in Aglets are not CORBA-based. In fact, they are defined as normal Java classes, interfaces or abstract classes that act as common wrappers for the protocols actually being used.

Unlike normal Java objects, aglets are never garbage-collected automatically, because an aglet is active and has its own threads of control. An aglet programmer needs to explicitly dispose of an aglet.

When an aglet has been dispatched, deactivated, or disposed of, the AgletRef object is removed from the reference table. In addition, the internal reference to that aglet and associated components such as MessageManager object or properties are set to null so that the garbage collector can sweep up these dangling objects. This means that if you have a live reference to this aglet elsewhere, it will not be Garbage Collected.

5 Mitsubishi's Concordia

Concordia is a full-featured framework for the development and management of network-efficient mobile agent applications that extend to any device supporting Java. Concordia consists of multiple components, all written wholly in Java, which combine together to provide a complete, robust environment for applications.

A Concordia System, at its simplest, is made up of a Java Virtual Machine (VM), a Concordia Server, and at least one mobile agent on 1 network node. Usually, there are many Concordia Servers, one on each of the various nodes of a network, both user and server nodes. The Concordia Servers are aware of one another and connect on demand to transfer agents in a secure and reliable fashion. The agent initiates the transfer by invoking the Concordia Server's methods. This signals the Concordia Server to suspend the agent and to create a persistent image of it to be transferred. The Concordia Server inspects an object called the Itinerary, created and owned by each agent, to determine the appropriate destination. That destination is contacted and the agent's image is transferred, where it is again stored persistently before being acknowledged. In this way the agent is given a reliable guarantee of transfer.

After being transferred, the agent is queued for execution on the receiving node. This happens promptly but possibly subject to certain administrative constraints. When the agent again begins executing, it is restarted on the new node according to the method specified in its itinerary, and it carries with it those objects which the programmer requested. Its security credentials are transferred with it automatically and its access to services is under local administrative control at all times.

The Concordia Server is a Java program which runs in the Java VM on machines in the network where mobile agents may need to travel. The Concordia Server is responsible for providing all Concordia functionality on a given machine. The Concordia Server manages the life cycle of the agent. It provides for agent creation and destruction, and provides an environment in which the mobile agent executes.

The Concordia components are:

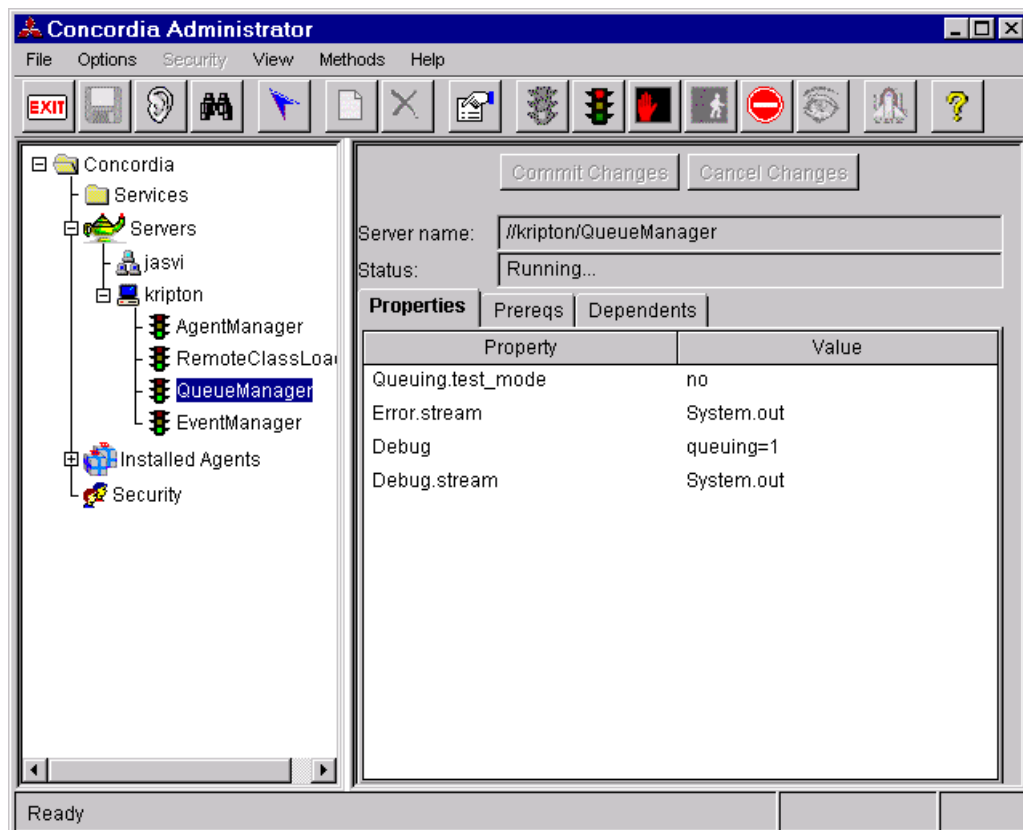
- The Agent Manager provides the communications infrastructure that allows for agents to travel. It abstracts the network interface so that agent programmers do not need network specifics or programming interfaces.
- The Security Manager protects resources and ensures the security and integrity of mobile agents and their data. Concordia security can be configured via a graphical user interface.
- The Persistence Manager maintains the state of mobile agents and objects in transit around the network. It makes it possible to restart mobile agents in the event of a server failure and restart.
- The Inter-Agent Communication Manager handles the registration, posting and notification of events to and from mobile agents. It provides for multicast events (send events to multiple recipients) and is distributed such that the sender and receiver of an event need not be on the same machine. Also, the Inter-Agent Communication Manager provides an infrastructure which allows the mobile agents to collaborate (i.e. synchronise and share data with each other).
- The Queue Manager is responsible for the scheduling and guaranteed delivery of mobile agents between Concordia servers. It provides for reliable transmission in an unreliable network.
- The Directory Manager provides a name service for applications and agents. It allows agents to find services in the network.
- The Administration Manager provides remote administration of Concordia. Only one
- Administration Manager is required in the Concordia System. The Administration Manager supports simultaneous, central administration of multiple servers. The Administration Manager has a user interface component that is its primary means of use.
- The Agent Tool Library is the set of development tools provided by Concordia. This includes all Concordia APIs (Administration APIs, Lightweight Agent Transport APIs, Service Bridge API, etc.) and agent classes needed to develop Concordia mobile agents.

As one might expect from the names of its various components, the Concordia system provides some features that are completely absent from the other agent systems. The Administration Manager provides a user interface for administering all the services provided by Concordia, including security, event handling and agent migration. Only one administrator is required per Concordia network.

Following are some of the features provided by Concordia.

- Concordia employs existing TCP/IP communications services. Concordia does not impose a protocol or distributed computing service of its own.
- Advanced management functions allows thousands of mobile agents to run on a single workstation. Concordia administration can start, stop, suspend, and resume Concordia Servers; view, stop, suspend, and resume agents at a Concordia Server; create, modify, delete users and/or permissions; upgrade and install Concordia Servers, monitor Concordia Server performance, and manage the components.
- Collaboration provides a number of benefits, such as enabling parallel operation over multiple servers or multiple networks. Using collaboration, an application can divide a task into subtask, the subtask can be carried out in the most appropriate places. The results of these sub-tasks are then assembled by the collaboration framework. A decision is made based upon the results, which can be used to determine destination, action, or other appropriate behaviour.
- The Service Bridge allows a developer to add services to a Concordia Server. Service Bridges may be managed remotely via the Concordia Administration Manager. For example, you can provide access to an application-specific service so the service does not need to travel with the mobile agent
- The Service Bridge also provides a way out of the Virtual Machine to the outside world.
- Persistence and Queuing provides for automatic retries of agent transmission and queue storage recovery in case of server and/or network failures. These 2 features also provide for load balancing when machines in a network provide different response time and the order of execution is important.
- The Itinerary specifies where a mobile agent travels. It provides a method to allow destinations to be added or removed either by the application, mobile agent or the Concordia Administration.
- Service Naming is a name service for applications and agents. In an environment where information is dynamic (i.e. the Internet), this provides an easy way to establish a list of locations where services reside.
- The Concordia Security Structure unlike most agent systems provides security based on the rights of the user of the applications - not the permissions given by the developer of the application. This provides for more control of which files, databases, resources, etc., are available to a specific end user. In addition, the security system protects resources from access by unauthorised mobile agents and protects mobile agents from being tampered with by unauthorised users.
- The Lightweight Agent Transporter API allows the developer to embed within a client application the ability to receive, execute, and launch Concordia Agents. The application can receive notifications from the mobile agent and can directly interact with mobile agent.

- Encryption is not technically a part of the Security. Concordia can provide Encryption as a security measure or the developer can plug in their own encryption scheme.

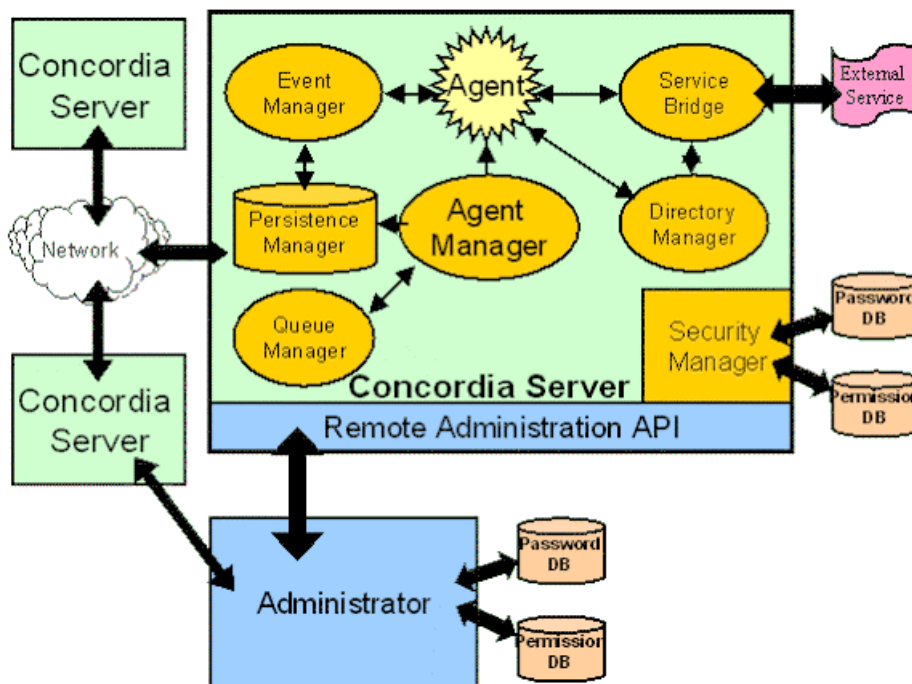


Concordia has recently finished the version 1.1.1 of the system, and we analyse a demo version without the security features. Concordia system comes with some utilities for:

- **Concordia Administrator:** a graphical user interface from which you can configure the Concordia network. You can obtain performances of the different services and managers
- **Concordia Agent Launch Wizard:** with this utility, you can send an agent to any Concordia server, specifying the method that must be executed when the agent arrives to the destination. You can also specify an itinerary for the agent.
- **Concordia Persistent Store Browser:** a graphical tool to see the objects stored in one server, with information about the memory occupied by the object, when it was stored, the OID and the result of applying the method toString() to the Object. It is a simple tool that makes it possible to look at and manipulate the contents of persistent store files. Persistent Store files are created and maintained by various Concordia components using the COM.meitca.concordia.persist.PersistentStoreManager API.

- **Concordia Queue File Browser:** is a tool for viewing and editing the contents of Incoming and Outgoing queue storage files. Queue files are created and used by the QueueManager component of the Concordia Server (if Queuing is enabled) to provide for reliable agent transmission.
- **Concordia Server Control Panel:** on 32-bit Windows platforms, the Concordia Server may be controlled and configured using the Concordia Server Control Panel. After installation of the Concordia kit, the Control Panel will start automatically each time a user logs in to the system. It may also be started manually using the Concordia Server Control Panel icon in the Start menu. The Concordia Server Control Panel requires Administrator privilege on Windows NT. It will not start automatically for a non-privileged user, and if a non-privileged user attempts to start it using the Start menu, an error message will be displayed and the Control Panel will exit. This behaviour is required to prevent non-privileged users from affecting the Server in a secure environment.

The Concordia Security management can be illustrate with the following diagram:



Under normal circumstances, each Concordia Server maintains its own local copy of the Security Passwords and Security Permissions databases. The Administrator runs in a separate process using a separate Java Virtual Machine than the Concordia Servers. The Administrator uses RMI to control all Concordia Servers, using a private remote Administration method interface. The Administrator itself accesses the Security Passwords and Security Permissions file of the Concordia Server to which it is "logged in".

Normally, the Administrator is logged into the local host, in which case the security files are read and written directly using normal file I/O. However, the Administrator may be logged into a remote server, in which case the security tables are accessed via a special

RMI method interface to the remote Server. In order to login to a remote Server, the Concordia Server must be running on the remote machine.

Each time the Administrator requests a remote operation, it sends the security ID of the user who logged into the Administrator to the remote Server. If the security ID is not valid or does not have sufficient privilege to perform the requested operation, an exception is thrown and the Administrator will report the error. The implication of this design is that in order for a user to use the Administrator to affect remote Servers, the User ID (including the password) must be identical on all Servers, and must have sufficient permissions define on it to enable the operation.

Concordia provides a complete set of resource permissions to govern access to the remote administration capabilities of Concordia. This makes it possible, for example, to set up an environment where some users are permitted to monitor the Concordia network, but not to change anything, while other users may be granted complete access to control all features of the system.

The list of security resources that can be controlled is:

- **AgentAdmin.Monitor**: this resource permission may be used to grant or deny the ability of an Administrator user to monitor agents on a Concordia Server using the Performance Monitor on the Agent Manager.
- **AgentAdmin.Manage**: this resource permission grants (or denies) the ability for a user to affect the execution of an agent on the Server using the AgentAdmin panel of the Performance Monitor.
- **PropertyAdmin.View**: this resource grants or denies the user access to examine the properties of the Concordia Server or any of its installed Services. These properties normally are displayed in the right pane of the Administrator window when the Server or a Service are selected.
- **PropertyAdmin.Modify**: this resource grants or denies the user the ability to modify the properties of the Concordia Server or any of its installed Services.
- **SecurityAdmin.Modify**: this resource grants or denies permission for the user to modify the password or permissions files on the Server or on the local host.
- **ServerAdmin.Monitor**: this resource grants or denies permission to view the performance statistics of a Concordia Server or any of its Services.
- **ServerAdmin.Manage**: this resource grants or denies permission to affect the execution or state of any of the installed Services on a Concordia Server.
- **ServerManagerAdmin.Monitor**: this resource grants or denies permission for a user to query a Concordia Server to determine which Services are installed on the Server, and their current state.
- **ServerManagerAdmin.Manage**: this resource grants or denies permission to modify the state of the Concordia Server or any of its Services, and also grants or denies permission to install or remove Services from the Server.

I was not able to examine the security issues, because I only could obtain a demonstration program, with all the features except the security part.

To develop mobile Concordia agents, you have several possibilities. Normally, you will extend the Agent class, which has all the methods need for an agent to be executed and to travel through a computer network. An Agent's travels are specified by its Itinerary, which is composed of a list of Destinations. Each Destination indicates the name of a machine on the network to which the Agent should travel, and the name of a method of the Agent that should be executed when the agent arrives at that host.

An AgentTransporter provides a lightweight infrastructure for launching, receiving, and executing Concordia Agents, without the use of a full Concordia Server. Typically, a Java application will create an instance of AgentTransporter, instead of starting a Concordia Server, to enable Concordia Agents to travel to/from itself and to provide and execution context for the agents. After creating an AgentTransporter, a Concordia Agent can travel into the application or applet, have one of its methods invoked by the AgentTransporter as specified by the Itinerary, and then continue travelling to other hosts. New agents can now also be created and launched.

From an applet, you cannot start an AgentTransporter. You have then a ConcordiaApplet that provides lightweight infrastructure for sending/receiving/executing Concordia Agents from within a Java applet.

A CollaboratorAgent is a subclass of Agent that enables collaboration among agents in a group. Agents wishing to collaborate must extend this class to do some useful work. They must also belong to at least on agent coordination group (AgentGroup). Generally, an application first creates an AgentGroup. Subsequently, it creates new CollaboratorAgents and populates the group with them.

A Secure Concordia Agent must be created extending the SecureAgent class. This class allows an agent to properly identify itself to the ConcordiaSecurityManager. SecureAgents carry three additional pieces of information with them.

- **Identification:** the SecureAgents Identification is the identity of the person using the SercureAgent. The SecureAgents Identification is used by the ConcordiaSecurityManager to determine what resource permissions the SecureAgent has access to on the Concordia Server.
- **ClientRestrictions:** prevent a SecureAgent from accessing resources on a Concordia Server. ClientRestrictions will prevent resource access regardless of the permissions assigned to the user of the SecureAgent. ClientRestrictions may be used by a developer to ensure that SecureAgents, derived from some SecureAgent the developer creates, can not access a certain resource. It may be that access to this resource will cause the developers Agent to behave improperly.
- **SecurityPermits:** they are used by the Concordia Server to temporarily assign a SecureAgent access to resource permissions required by the Concordia Server. Server Permits provide a way for the Concordia Server code to ensure that a SecureAgent has permission to access a particular resource.

There is also the SecureCollaboratingAgent to create secure agents that collaborate within an AgentGroup.

6 General Magic's Odyssey

With the advent of Java, General Magic began developing Odyssey, a new agent system implemented solely in Java that incorporates some of the concepts previously developed for Telescript. Actually, they are in the beta 2 release.

Odyssey uses Java RMI. It also supports CORBA and DCOM protocols for agent transport. A rmiregistry process must be started in every machine on which an Odyssey agent server will run. Odyssey will attempt to start a rmiregistry process if one is not running when it is needed. A limitation with the Odyssey server is that only one can run per machine because the name scheme it uses for agent contexts.

Odyssey is an agent system implemented as a set of Java class libraries that provide support for developing distributed mobile applications. Odyssey provides Java classes for agents and places. Odyssey agents are Java threads. They are created by subclassing the Odyssey agent class or the Odyssey worker class.

The Odyssey Worker class is a subclass of the Odyssey Agent class. A worker is structured as a set of tasks and a set of destinations. At each destination, the worker executes to completion the next task on its task list.

An Odyssey worker may manipulate its task list at any point during its travels. For example, a worker could go to a Yellow Pages place, find the top suppliers of rare books, and add these places as new destinations and tasks in the task list.

The Odyssey agent system consists of a set of Java classes to support Odyssey agents and Odyssey places.

A place is a context within an agent system in which agents execute. A mobile agent travels between places. This context can provide functions such as access control. A place is the stationary part of the application you write. Because agents execute only within places, you add an instance of one or more places to each host running your application.

In Odyssey, the initial place created when the system is started is distinct from all other places. This place is derived from the class BootPlace. When the BootPlace exits, the Odyssey agent system shuts down, terminating all agents and places currently existing within that agent system.

The place provides the interface to the functionality that is local at each host running your application. This local functionality might include, the user interface code, and access to the local databases. In general, a place is the gateway between agents visiting a host and the host's resources.

The Odyssey class hierarchy contains the classes that implement the Odyssey paradigm. There are two classes for constructing mobile agents: Agent and Worker. There is also a class, called Place, for the agent execution environments.

The hierarchy also includes classes that support agents, workers, and places. These classes include Ticket (specifies how and where an agent travels), Means (specifies how an agent travels), Petition (identifies who an agent wants to communicate with), and ProcessName (used to generate the unique names of all processes, including agents and places).

The class hierarchy also includes three interfaces: genmagic.odyssey.AgentSystem, genmagic.odyssey.Finder, and genmagic.odyssey.Transport. These interfaces allow a developer to customize the implementation of an Odyssey agent system.

7 Objectspace's Voyager

ObjectSpace Voyager is 100% Java and is designed to use the Java language object model. Voyager allows you to use regular message syntax to construct remote objects, send them messages, and move them between programs. The root of the Voyager product line is the ObjectSpace Voyager Core Technology. This product contains the core features and architecture of the platform, including a full-featured, intuitive object request broker (ORB) with support for mobile objects and autonomous agents. Also in the core package are services for persistence, scalable group communication, and basic directory services. Voyager Core Technology is actually for free. There is another version, Voyager Pro, that will be available by the end of 1998, which implements for security control and many other features.

You can download the Voyager Core Technology for its web site:
<http://www.objectspace.com/products/voyager/core/index.html>

In the installation package there are the voyager utilities (voyager, igen and cgen), the documentation files, examples and the Voyager jar file. The documentation included contains the Voyager API guide (html) and the Voyager Core Technology User Guide (pdf).

In the Objectspace web site, there is some other extra information about voyager:

- ObjectSpace Voyager™ Version 2.0.0 User Guide
- Voyager 2.0.0 Production API Documentation
- Voyager JDBC Activation Example
- Voyager PSE Activation Example
- Secure Java Applications Using Voyager and SSL
- ObjectSpace Voyager Technical Overview
- ObjectSpace Voyager and RMI Comparison

- ObjectSpace Voyager Agents Comparison
- ObjectSpace Voyager CORBA Integration Technical Overview
- ObjectSpace Voyager Transaction Service Technical Overview

Here is the list of Voyager features:

- **Remote-Enabling a Class:** Java classes are remote-enabled classes at runtime. A class does not have to be modified in any way, and no additional files are created.
- **Construction:** you can create a remote instance of any class and obtain a proxy to the newly created object. The proxy implements the same interfaces as the created object, and the proxy class is generated dynamically if it doesn't already exist.
- **Dynamic Class Loading:** classes can be dynamically loaded from one or more locations when necessary. This allows you to easily set up class repositories that serve your corporate Java applications.
- **Remote Messaging:** method calls made to a proxy are forwarded to its object. If the object is in a remote program, the arguments are serialized using the standard Java serialization mechanism and deserialized at the destination. The morphology of the arguments is maintained. By default, parameters are passed by value. However, if an object's class implements `IRemote` or `java.rmi.Remote`, the object is passed by reference instead.
- **Exception Handling:** if a remote exception occurs, it is caught at the remote site and rethrown locally.
- **Distributed Garbage Collection:** The distributed garbage collector reclaims objects when there are no more local or remote references to them. It uses an efficient "delta pinging" algorithm that keeps the traffic required for garbage collection to a minimum.
- **Dynamic Aggregation™:** This feature allows you to add secondary objects (termed *facets*) to a primary object at runtime. For example, you can dynamically add hobbies to an employee, a repair history to a car, or a payment record to a customer. Dynamic aggregation represents a fundamental step forward for object modelling and complements the traditional mechanisms of inheritance and polymorphism.
- **CORBA:** There is full native support for IDL, IIOP, and bidirectional IDL<->Java translation. No stub generators or helper classes are required.
- **Mobility:** You can move any serializable object between programs at runtime. If a message is sent from a proxy to an object's old location, the proxy is automatically updated with the new location and the message is resent. Mobility is often useful when optimizing message traffic in a distributed system.
- **Autonomous Mobile Agents:** you can create mobile autonomous agents that move themselves between programs and continue to execute upon arrival. It is easy to build agents that use movement to more efficiently satisfy their goals.
- **Activation:** the activation framework allows objects to be persisted to any kind of database and automatically re-activated in the case that the program is restarted. An object does not have to be modified in any way to be activable.

- **Applets and Servlets:** it is easy to create Voyager-enabled applets and servlets. Because applets cannot open network connections to any machine except their server, Voyager allows you to set up a server-side hub that can perform message routing and dynamic proxy generation on the applet's behalf.
- **Naming Service:** the naming service provides a single, simple interface that unifies the many commercially available naming services. New naming services can be dynamically plugged into Voyager's naming service.
- **Multicast:** you can multicast a Java message to a distributed group of objects without requiring the sender or receiver to be modified in any way.
- **Publish-Subscribe:** you can publish a Java event on a specified topic to a distributed group of subscribers. The publish-subscribe facility supports server-side filtering and wildcard matching of topics.
- **RMI:** the semantics of RMI's Remote interface and RemoteException class are supported. This means that you can easily use classes in Voyager that were originally designed for use with RMI.
- **Timers:** a Stopwatch and Timer class facilitate common timing chores. Timer events can be distributed and multicast if necessary.
- **Thread Pooling:** a thread pool is used when allocating and deallocating threads, resulting in higher performance.
- **Advanced Messaging:** you can send oneway, sync, and future messages. Oneway messages return immediately and discard the return value. Future messages immediately return a placeholder to the result, which may then be polled or read in a blocking fashion.
- **Security:** an enhanced security manager is included, as well as hooks for installing custom sockets such as SSL.

Voyager Server

There are several ways to start a voyager server. Included with the installation package, there is an executable program that starts a voyager server from the command line. This utility creates an empty Voyager program that accepts objects and messages from other Voyager programs.

On Windows NT it is also possible to install the server as a system Service, using the tool JService, that can be found at <http://www.bmobile.com/JService>.

From a program, if you want to use any Voyager feature you have before to start the server using the `Voyager.startup()` method. There are different methods to start the server:

- `Voyager.startup()`: startup as a client
- `Voyager.startup(null)`: startup as a server on a random unassigned port
- `Voyager.startup("8000")`: startup as a server on port 8000
- `Voyager.startup("//dallas:7000")`: startup as server on port dallas:7000

To shutdown the server, invoke the method `Voyager.shutdown()`.

By default, when you start a Voyager server, you don't install any security manager. You have to explicitly install a Java Security Manager using the `System.setSecurityManager()` method. The security in Voyager will be explained later.

Agents and Remote Objects

Having objects that can move from one machine to another for exchanging a large number of messages closer to the other objects reduces network traffic and increase throughput. Local messages are often 1000 times faster than its remote equivalent. Also, having local communication between objects, hosts don't need to be connected to the network.

In the last release of Voyager, you don't need to extend the `Agent` class to create a new Agent. Any class will be converted in agent by calling the method `Agent.of(<class>)`, which return an interface object that contains the methods to work with agents (`getHome()`, `isAutonomous()`, `moveTo()` and `setAutonomous()`).

Any serializable object can become a mobile agent using the Voyager dynamic aggregation feature (explained below). When you first call the `Agent.of()` method, you create a new facet in the object, which is the `IAgent` interface that contains the methods to enable the object to be mobile.

Voyager also offers the possibility of remote object creation. To create an object at a specific location, you have to use the `Factory.create()` method. This method returns a proxy to the newly created object, and creates the proxy class dynamically if it does not already exist.

There are several variations of `create()`, depending on whether the object is to be created locally and whether the class's constructor takes arguments. The name of the class must always be fully qualified. To create a class locally in the program, you only need to specify the fully qualified name of the class (e.g. `java.util.Vector`). To create the class remotely, you also have to specify the name of the remote machine and the port. If the constructor of the class takes any argument, you can also pass them when you call to the `create()` method.

You can move an object to a new location using the `Mobility.of()` method to obtain a object's mobility facet, and then using the `moveTo()` method defined in `IMobility`.

When you call the `moveTo()` method, any message that the object is currently processing is allowed to complete and any new messages that arrive at the object are suspended. The code that does this can only detect method calls that are synchronized, so if you must take care of do not move an object that is executing non-synchronized methods. The object and all of its non-transient parts are copied to the new location using Java serialization. To avoid copying a particular part, you must declare it as a proxy.

In the system where the object resides, Voyager keep a reference to the new location, and any message received in its old location will be forwarded to the new one, and the proxy that references to the old address will be update for news messages.

An object can be notified before been moved. If the object or any of its parts implements the IMobility interface, then it will receive callbacks during a move in the following order:

- **PreDeparture()**: this method is executed on the original object at the source. If the method throws a MobilityException, the move is aborted.
- **PreArrival()**: this method is executed on the copy of the object at the destination. If the method throws a MobilityException, the move is aborted.
- **PortArrival()**: at this point, the copy of the object has become the real object, the object at the source has become the stale object, and the move is deemed successful and cannot be aborted. `postArrival()` is executed on the copy of object at the destination immediately prior to the user-supplied callback, and is typically defined to perform activities such as adding the new object into persistent storage.
- **PostDeparture()**: this method is executed on the original stale object at the source, and is typically defined to perform activities such as removing the stale object from persistence. Messages sent to the stale object via a proxy will be redirected to the new object, so `postDeparture()` should not utilise proxies to the original object or any of its facets.

Sending Messages and handling Exceptions

You can send synchronous messages in Voyager using regular Java syntax. For other types of messages, Voyager provides a message abstraction layer. You can dynamically construct messages and send them to any remote object or agent. There are three different kind of messages:

- **Synchronous**: using regular Java syntax or `Sync` (dynamically).
- **Asynchronously**: using `Oneway` or `Future` messages.

A message sent via a proxy is executed according to the following rules:

- If the destination object is in the same program, the message is delivered just like a regular Java message. The arguments are not serialized or copied, resulting in very high performance.
- If the destination object is in a different program, the arguments and return value must be sent across the network. If an argument implements `com.objectspace.voyager.Iremote` or `java.rmi.Remote`, a proxy to the argument is sent (pass by reference), otherwise a copy of the argument is sent using standard Java serialization (pass by value). Morphology of the arguments is maintained: an object that is an argument or part of an argument is copied exactly once, and an argument or part of an argument that shares an object in the local program will also share a copy of the object in the remote program. The rules described for an argument also apply to a return value.

By default, Voyager messages are synchronous. When a caller sends a synchronous message, the caller blocks until the message completes. Synchronous messages can be sent using regular Java syntax, but you can also send synchronous messages dynamically using the `Sync.invoke()` method, which returns a `Result` object. This method has three arguments:

- The target object
- The name of the method you want to call on the target object. If there are more than one method with the same name, you can also specify the argument typed using the syntax `method(type1, type2)`.
- The parameters to the invoked method in an object array.

You can query the Result object using the following methods:

- **isAvailable()**: return true if the Result object has received its return value.
- **readXXX()**, where XXX = Boolean, Byte, Char, Short, Int, Long, Float, Double, Object: return the value of the result object, blocking until either the value is received or the timeout period elapses. If you attempt to call the read method and a remote exception was thrown, it will be automatically rethrown.
- **isException()**: wait for a reply and return true if Result contains an exception.
- **getException()**: return the exception contained in Result, or null if no exception occurred.

A One-Way message is a message that does not return a result. When the caller sends a one-way message, it doesn't block while the message completes.

If a remote method throws an exception, it is caught and re-thrown in the local program. If a Voyager-related exception occurs and the interface method explicitly throws `java.rmi.RemoteException`, the exception is thrown wrapped in a `RemoteException`, otherwise it is thrown wrapped in a `com.objectspace.voyager.RuntimeRemoteException`. In both cases, the public detail field contains the original exception.

Also, Voyager has a Console class that allows you to log information, including traces of remote exceptions, to the console. There are three different logging levels:

- **Silent**: display no output in console.
- **Exceptions**: display stack traces of remote exceptions and unhandled exception in the console.
- **Verbose**: display stack traces of remote exceptions, unhandled exceptions and internal debug information in the console.

Multicast and Publish/Subscribe feature

Most traditional systems use a single repeater object to replicate a message or event to each object of a group. This approach works fine with a small number of objects, but does not scale well when large numbers of objects are involved. Voyager uses a scalable architecture for message/event replication called Space.

A Space is a distributed container of objects and can span multiple programs. A Space is created by linking together one or more Subspaces. A Subspace is a local container of objects.

A message/event sent via multicast proxy into a Subspace is cloned to each of its neighbouring Subspaces before being delivered to every object in the local Subspace. As

the message propagates, it leaves a marker unique to that message that is remembered for a period of five minutes. If a clone of that message arrives to the Subspace, the clone detects the marker and self-destructs. Subspaces can be connected using an arbitrary topology, so the Space becomes more fault-tolerant in the face of individual network failures.

In a Subspace, you can use the method `connect()` to connect the Subspace with any other Subspace (connections are symmetric), and `add()` to add objects to a Subspace. You also have methods to disconnect, get a list of the neighbours, remove object from a Subspace, get a list of the objects contained in a Subspace and check if an object is contained in a Subspace.

Voyager allows you to multicast a Java message to a group of objects in a Space. There are to methods that allow you to do that:

- **Multicast.invoke**(String methodName, Object[] args, String classname): send a one-way message to every object in the Space that is an instance of the specified class or interface.
- **Subspace.getMulticastProxy**(String classname): return a multicast proxy that is type compatible with the specified class or interface. Messages sent to this proxy are sent to all the objects in the Space that are an instance or implement the specified class or interface.

If you wants to publish an event associated with a topic to every object that implements `PublishedEventListener` in a Space, use `Subspace.publish(event, topic)`.

`PublishedEventListener` defines a single method `publishedEvent(event, topic)` that receives every published event in the Space. It is up to the listener to handle the event in the appropriate manner.

Naming Service

A naming service allows names to be associated with an object for later lookup. There are many different implementations of naming services: Voyager federated directory service, CORBA naming service, JNDI, Microsoft Active Directory, RMI registry.

Each naming service adds a prefix for later identification (*vdir* for Voyager federated directory, *IOR* for CORBA).

8 Other agent systems in Java

8.1 Java Agent Template

The JAT provides a fully functional template, written entirely in the Java language, for constructing software agents that communicate peer-to-peer with a community of other

agents distributed over the Internet. Although portions of the code that define each agent are portable, JAT agents are not migratory but rather have a static existence on a single host. This behavior is in contrast to many other "agent" technologies. All agent messages use KQML as a top-level protocol or message wrapper. You can find the current KQML standard at <http://www.cs.umbc.edu/kqml/>.

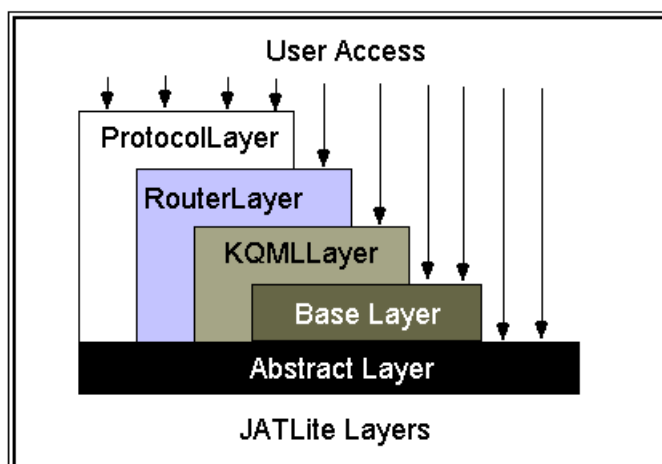
JAT agents can be executed as either standalone applications or as applets via the appletviewer. Coordination is provided by an Agent Name Server. The architecture of the JAT was specially designed to allow for the replacement and specialization of major functional components including the GUI, low-level messaging, message interpretation and resource handling.

The JAT package is in the version 0.3, and will no longer be supported by the creators (University of Standford). They have now a new package called JATLite (Java Agent Template, Lite).

JATLite provides a template for building agents that utilise a common high-level language and protocol. This template provides the user with numerous predefined Java classes that facilitate agent construction. Furthermore, the classes are provided in layers, so that the developer can easily decide what classes are needed for a given system. For instance, if the developer decides not to use KQML, the classes in the KQML layer are simply omitted.

It does provide a robust substrate for building such intelligent agents. The JATLite packaged infrastructure allows agents to be portable (e.g., on a laptop computer), to move from one machine to another, and to connect and disconnect from the Internet with automatic queuing and buffering of incoming messages. These features, found to be necessary for robust agent behaviour in projects where software agents occasionally fail or migrate, are provided by the Agent Message Router (AMR) infrastructure.

The architecture of JATLite is organised as a hierarchy of increasingly specialised layers, so that developers can select the appropriate layer from which to start building their systems. Thus, a developer who wants to utilise TCP/IP communications but does not want to use KQML can use only the Abstract and Base layers as described below.



JATLite can be found at <http://java.stanford.edu/>. Actually there is a beta version for download, and also, some examples of how to use JATLite and documentation about the package.

8.2 NCSA Habanero

NCSA Habanero is a collaborative framework and set of applications. Using Habanero you can create and work in shared applications from remote locations over the Internet. The Habanero framework, or API, enables developers of groupware applications to build powerful collaborative software in a reduced amount of time. The Habanero framework provides the necessary methods developers can use to create or convert existing applications into collaborative applications. Habanero is written in Java, it will run under any operating system that supports JDK 1.1.6. The Habanero environment consist of a client, a server and a variety of tools

Habanero works by replicating applications across clients and then sharing all state changes in those clients. When a new client joins a session, it is sent information about which applications are running in that session. Each application is then sent enough information to completely replicate the important state being shared by the existing copies of that application. Habanero also ensures that all clients see the same state changing events in the same order, which results in applications appearing the same to all clients.

The NCSA Habanero can be found at <http://www.ncsa.uiuc.edu/SDG/Software/Habanero/>.

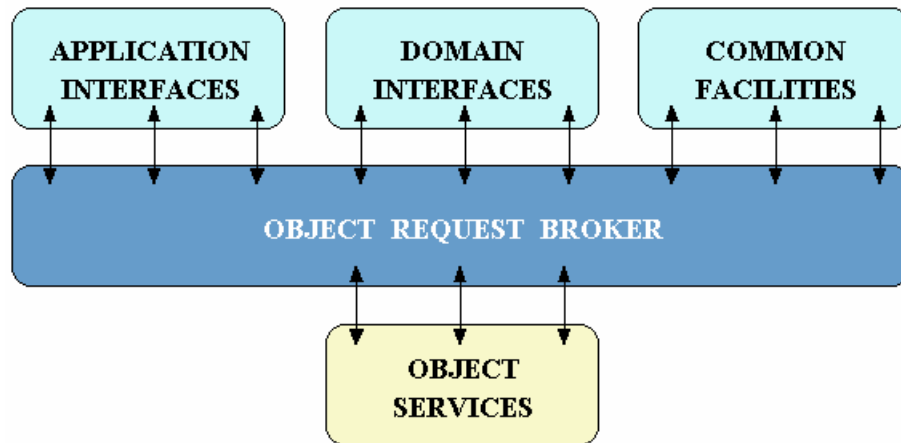
9 Other techniques for distributed computing

There are some other techniques that can be used instead of/with the agent paradigm. All of them are implemented in Java, but can also be used from C++, or from the web. Four different methods are explained here, but there are many others. As Java is a recent technology, and there are a lot of people working around, new features, extensions and updates keep going out very often.

9.1 The Common Object Request Broker (CORBA)

The Common Object Request Broker Architecture is an emerging open distributed object computing infrastructure being standardized by the Object Management Group (OMG). CORBA automates many common network programming tasks such as object registration, location, and activation; framing and error handling; and operation dispatching. See the OMG Web site (<http://www.omg.org/>) for more overview material on CORBA.

The following figure illustrates the primary components in the OMG Reference Model architecture.



- **Object Services:** these are domain-independent interfaces that are used by many distributed object programs. For example, a service providing for the discovery of other available services is almost always necessary regardless of the application domain. Two examples of Object Services that fulfil this role are:
 - **The Naming Service:** allows clients to find objects based on names.
 - **The Trading Service:** allows clients to find objects based on their properties.

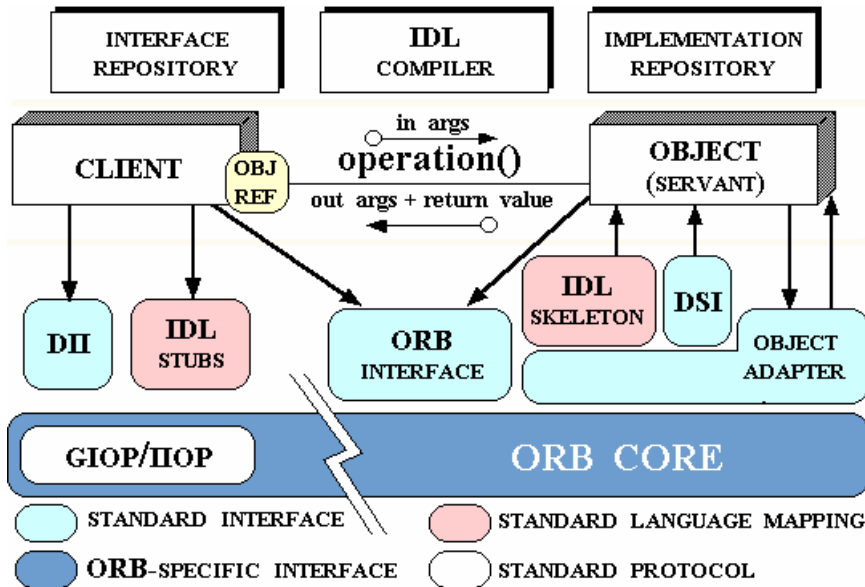
There are also Object Service specifications for lifecycle management, security, transactions, and event notification, as well as many others.

- **Common Facilities:** like Object Service interfaces, these interfaces are also horizontally oriented, but unlike Object Services they are oriented towards end-user applications. An example of such a facility is the Distributed Document Component Facility (DDCF), a compound document Common Facility based on OpenDoc. DDCF allows for the presentation and interchange of objects based on a document model, for example, facilitating the linking of a spreadsheet object into a report document.
- **Domain Interfaces:** these interfaces fill roles similar to Object Services and Common Facilities but are oriented towards specific application domains. For example, one of the firsts OMG RFPs issued for Domain Interfaces is for Product Data Management (PDM) Enablers for the manufacturing domain. Other OMG RFPs will soon be issued in the telecommunications, medical, and financial domains.
- **Application Interfaces:** these are interfaces developed specifically for a given application. Because they are application-specific, and because the OMG does not develop applications (only specifications), these interfaces are not standardised. However, if over time it appears that certain broadly useful services emerge out of a

particular application domain, they might become candidates for future OMG standardisation.

CORBA ORB Architecture

The following figure illustrates the primary components in the CORBA ORB architecture:



- **Object implementation:** this defines operations that implement a CORBA IDL interface. Object implementations can be written in a variety of languages including C, C++, Java, Smalltalk, and Ada.
- **Client:** this is the program that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, i.e., `obj->op(args)`.
- **Object Request Broker (ORB):** the ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.
- **ORB Interface:** an ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting

object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

- **CORBA IDL stubs and skeletons:** CORBA IDL stubs and skeletons serve as the “glue” between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler.
- **Dynamic Invocation Interface (DII):** this interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking deferred synchronous (separate send and receive operations) and oneway (send-only) calls.
- **Dynamic Skeleton Interface (DSI):** this is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.
- **Object Adapter:** this assists the ORB with delivering requests to the object and with activating the object. An object adapter associates object implementations with the ORB. Object adapters can be specialised to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).

Voyager CORBA

Voyager includes CORBA integration. Due to its powerful Java-centric architecture, Voyager provides in many ways a better solution for CORBA-enabling a Java program than any other CORBA solution available today.

- Voyager can take any IDL file and automatically create its equivalent Java interface and Voyager virtual reference class. It can also take any Java interface or .class file and automatically create its equivalent IDL file, allowing a Java class to be CORBA-enabled without modification.
- Voyager can obtain a virtual reference to a CORBA object, and CORBA can obtain a remote reference to a Voyager object.
- A Voyager developer can send messages to an object via a virtual reference without knowing whether the object is a CORBA object or a Voyager object. If the object is in a CORBA ORB, Voyager automatically uses IIOP to communicate with the CORBA object.
- Virtual references to Voyager objects are automatically converted to CORBA references when sent to a CORBA ORB, and CORBA references are automatically converted to virtual references when sent to Voyager.

- Voyager supports in, inout, and out parameters via the standard holder mechanism.
- Because virtual references hide the details of the underlying ORB, Voyager's advanced services, such as Space™, multicast messaging, futures, and dynamic invocation, can be used with CORBA objects, Voyager objects, or a mixture of both CORBA and Voyager objects.

Many existing classes can be used directly in a CORBA program without modification. As usual, proxy classes are generated at runtime, so no stub generator is required. However, there are four limitations of CORBA that affect the classes that can be used within a CORBA program.

- CORBA does not allow two or more methods in an interface to have the same name. If an existing Java class has duplicate method names, you must rename the methods so that they do not clash.
- CORBA does not support pass-by-value. When you pass an argument to a CORBA method, it is always sent as a proxy. If the argument is not already a proxy, Voyager automatically converts it into a proxy using `Proxy.of()`.
- CORBA does not support distributed garbage collection. If you pass a proxy to a remote CORBA program, the remote proxy will not prevent the garbage collection of the original object. By default, Voyager disables the garbage collection of objects that are automatically converted into CORBA proxies by anchoring them to the local VM. Anchored objects are never garbage collected. To change this setting, you can use `Corba.setAnchoring()`.
- CORBA does not support inheritance of exceptions. All exceptions that are intended for use with CORBA should directly extend `java.lang.Exception`, declare all their data members as public, and include a public constructor whose first argument is the exception message and the remaining arguments match each of the public data members. The exception constructor should propagate the exception message argument up to the Exception class.

The simplest way to import/export a CORBA object is by using IORs (Interoperable Object References). An IOR is a string that encodes the host name, port number, type, and key of a single CORBA object.

To obtain an object's IOR, pass the object or a proxy to `Corba.asIOR()`. If the argument is not already a proxy, Voyager automatically converts it into a proxy using `Proxy.of()` and anchors the original object into the local VM to prevent garbage collection.

A server can export an object to a client by writing its IOR to a file so that the client can read the IOR and bind to it. A client can obtain a proxy to a CORBA object by passing its IOR to `Namespace.lookup()`.

Another way to import/export CORBA objects is via a CORBA naming service. Once you have obtained an IOR to a CORBA naming service and bound to it, you can use the standard CORBA naming service API to add and obtain references to CORBA objects.

Any message sent to a remote CORBA object is transmitted using IIOP. If an ORB-related exception occurs at any time during the remote method call, a runtime

CorbaSystemException is thrown. If a regular exception is thrown by the remote object, it is caught and then rethrown on the client side.

Voyager support the full range of IDL types:

- **struct**: pass a collection of data fields by value.
- **union**: pass a single field by value.
- **enum**: denote one of a small range of values symbolically.
- **array**: pass a statically-sized list of elements.
- **sequence**: pass a dynamically-sized list of elements.
- **any**: pass a single object of any IDL type.
- **typecode**: encode information about a particular IDL type.

You cannot create Java classes for these types directly. Instead, you must define the types in IDL and then use cgen to create their Java equivalents automatically. Use of these types closely follows the standard IDL to Java specification. Voyager supports recursive typecodes so that types may directly or indirectly refer to themselves.

The following table lists the mappings for primitives and strings between IDL and Java:

IDL	JAVA
boolean	boolean
char	char
wchar	char
octet	byte
short	short
unsigned short	short
long	int
unsigned long	int
long long	long
unsigned long long	long
float	float
double	double
long double	not supported
fixed	not supported
string	java.lang.String
wstring	java.lang.String

Voyager offers an utility to convert IDL files to and from Java. For more information about the Java to IDL or IDL to Java mapping, consult the appendix B of the Voyager documentation, or the “Java Language to IDL mapping” document from OMG.

9.2 RMI

Introduced in JDK 1.1 as one of the "Enterprise APIs," Remote Method Invocation (RMI) provides a communication transport mechanism between a Java client and a Java server.

RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Distributed object applications need to:

- **Locate remote objects:** applications can use two different mechanisms to obtain references to remote objects. It can register its remote objects with RMI's simple naming facility, the `rmiregistry`. Or the application can pass and return remote object references as part of its normal operation.
- **Communicate with remote objects:** details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard Java method invocation.
- **Load class bytecodes for objects that are passed around:** because RMI allows a caller to pass pure Java objects to remote objects, RMI provides the necessary mechanisms for loading an object's code as well as transmitting its data.

RMI uses a standard mechanism (employed in RPC systems) for communicating with remote objects: stubs and skeletons. A stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub that is responsible for carrying out the method call on the remote object. In RMI, a stub for a remote object implement the same set of remote interfaces that a remote object implement.

When a stub's method is invoked, it does the following:

- initiates a connection with the remote VM containing the remote object.
- marshals (writes and transmits) the parameters to the remote VM
- waits for the result of the method invocation
- unmarshals (reads) the return value or exception returned
- returns the value to the caller

The stub hides the serialisation of parameters and the network-level communication in order to present a simple invocation mechanism to the caller.

In the remote VM, each remote object may have a corresponding skeleton. The skeleton is responsible for dispatching the call to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following:

- unmarshals (reads) the parameters for the remote method
- invokes the method on the actual remote object implementation
- marshals (writes and transmits) the result (return value or exception) to the caller

RMI allows parameters, return values and exceptions passed in RMI calls to be any object that is serializable. RMI uses the object serialization mechanism to transmit data from one virtual machine to another and also annotates the call stream with the appropriate location information so that the class definition files can be loaded at the receiver.

For the garbage collector, the RMI runtime keeps track of all live references within each Java virtual machine. When a live reference enters a Java virtual machine, its reference count is incremented. When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference. The weak reference allows the Java virtual machine's garbage collector to discard the object if no other local references to the object exist. The distributed garbage collection algorithm interacts with the local Java virtual machine's garbage collector in the usual ways by holding normal or weak references to objects.

The RMI transport layer normally attempts to open direct sockets to hosts on the Internet. Many Intranets, however, have firewalls that do not allow this. The default RMI transport, therefore, provides two alternate HTTP-based mechanisms that enable a client behind a firewall to invoke a method on a remote object which resides outside the firewall.

The RMI also implements a Naming service. The `java.rmi.Naming` class provides methods for storing and obtaining references to remote objects in the remote object registry. The Naming class's methods take, as one of their arguments, a name that is formatted of the form: “//host:port/name”. Binding a name for a remote object is associating or registering a name for a remote object that can be used at a later time to look up that remote object. A remote object can be associated with a name using the Naming class's `bind` or `rebind` methods.

RMI and Voyager

The following table shows a comparison between Voyager and RMI:

Feature	Voyager	RMI
Constructing a remote object	Supported via regular Java syntax	Not supported
Remote-enabling a class	Requires on step	Requires five steps
Exporting a named object	Seamlessly integrated	Requires external registry
Connecting to a named object	Seamlessly integrated	Requires external registry
Exception handling	Explicit or run-time exceptions allowed	Only explicit exceptions allowed
Executing a remote static method	Supported via regular Java syntax	Not supported
Object mobility	Fully supported	Not supported
Agents	Can execute as they move and can move themselves	Not supported
Distributed persistence	Supported in Voyager 2.0 beta 2, and will be supported in Voyager2.1 (not supported in Voyager 2.0)	Not supported
Scalability	Distributed computing supported with Space architecture	Not similar feature
Multicast messaging	Fully supported, 100% non intrusive	Not supported
Distributed events	Compliant with JavaBeans	Not supported
Publish/subscribe	Messages and events supported	Not supported
Message types	Synchronous, one-way, future	Only synchronous messages
Evolution of classes	Supported	Not supported
Garbage collection	Lease- and time-based GC	Only lease-based GC
Applet connectivity	Unrestricted	Restricted
Network class loading	Built-in	Requires Web server

The table below lists a few benchmarks that compare RMI and Voyager performance on remote method calls between objects on the same virtual machine and between objects on different virtual machines. Each function was defined to take a specific kind of argument and to perform no operation. The benchmarks were performed on a 150Mhz Tecra laptop with 80MB of RAM. Times are in milliseconds per function call. The following interface definition was used.

```

package benchmarks;
import java.util.Vector;
import java.rmi.*;
public interface IServer extends Remote {
public void noArguments() throws RemoteException;
public int twoInts( int a, int b ) throws RemoteException;
public int vectorIntegers( Vector integers ) throws RemoteException;
public int vectorStrings( Vector strings ) throws RemoteException;
}

```

	No Arguments	Two Integers	Vector of 100 Integers	Vector of 100 Strings
Same virtual machine				
RMI	2.1	2.3	437.83	193.48
Voyager	0.2	0.5	0.3	0.4
Different virtual machines				
RMI	2.1	3.01	436.02	190.28
Voyager	3.0	3.21	117.87	193.98

9.3 Servlets

The original standard for server-side scripts is CGI or Common Gateway Interface. CGI is simple and widely supported. The main problem with CGIs is that it is not very efficient because the server launches one copy of the script per request and the overhead can be significant. Vendors have developed proprietary alternatives such as ISAPI (Microsoft) and NSAPI (Netscape) to address the performance issue. JavaSoft proposed the Java servlets as a standard efficient alternative to CGI.

Servlets are server-side Java programs that provide a means of generating dynamic Web content. They are not standalone applications that can be executed from the command line; instead, they run within Web servers. In order to run servlets inside a Web server the server must have a Java Virtual Machine running within itself. Unlike applets, servlets are not constrained by security restrictions. They have the capabilities of a full-fledged Java program and can access files for reading and writing, load classes, change system properties, etc. They are restricted only by the file system permissions, just like other Java application programs.

A servlet is loaded the first time it is used, and it remains in memory for later requests. It has an `init` method, where you can initialise the state of the servlet, and different methods for the different calls (`doGet`, `doPost`, `doPut`, `doDelete`). There is also a `destroy` method to manage the resources that are held by the servlet.

You can use a servlet to perform typical server-side processing. The servlet can communicate with the client computer and it can also communicate with other remote, networked computers.

An intelligent agent can be implemented as a servlet to monitor the health of your computer network. The servlets can poll host machines on your network at given intervals and ensure that some services are operating. The results can be stored in a database or displayed in a real-time graphical applet. In the event of an emergency, the servlet could send e-mail to the system administrator.

Java servlets have full access to Java's networking features. The servlets can connect with other networked computers using sockets or Remote Method Invocation (RMI). Also, the servlet can easily connect to an Objectivity/DB using the Objectivity Java bindings. The main restriction using Objectivity federated databases is that you can only work with one federated database. As the Java bindings from Objectivity/DB uses static methods keep the information about the federated database currently opened, only one federated database can be opened per virtual machine. But the servlets are all executed in the same VM, so you can only have one federation per server.

One possible solution for this restriction can be done with the help of agents. As said before, a servlet has no special restrictions, so it can begin a new VM, and start there an

agent server. Then the servlet can send there an agent that will open the federated database, and work with it. This solution let you open more than a federation using the same servlet, but you will have the same performance problem than using CGIs: start a new shell (in our case, a new VM) per request.


[Here](#) is an example of a servlet that shows the database information of a federated database. When you first call the servlet, it shows you a form and asks for the boot file of the federation. Then, the servlet try to open the federated database, and retrieve the information concerning the autonomous partitions, databases and containers of the federation. Once you specify a boot file, it will always show the information of the same federated database, even if you specify a different boot file later.

10 Agent systems comparison

This chapter makes a comparison of all the agent products presented until now: Aglets, Concordia, Odyssey and Voyager.

All the agent platforms presented here have a common set of features. The greatest differences are in the way of creating new agents, the communication between agents and the way of managing the servers. The two systems that offer more features and that are easier to learn and use are Objectspace's Voyager and IBM's Aglets.

The table shows the features offered by the agent platforms.

Feature	Aglets	Concordia	Odyssey	Voyager
Create agent	Locally	Locally	Locally	Locally and remote
Sending Java messages remotely	Not allowed	Not allowed	Not allowed	
Sending messages to mobile agents	Using the Message class	Inter-Agent Communication Manager using events	Petition class	Regular Java syntax
Message modes between agents	Synchronous Future One-way	Synchronous Multicast	Synchronous	Synchronous Future One-way One-way multicast
Life spans	Explicit deletion	Explicit deletion	Explicit deletion	When no more references (locals and remotes)
Directory service	Not included	Directory Manager	ProcessName	Naming service
Object Mobility	Not supported	Not supported	Not supported	Serialisable objects without any code modification
Agent Mobility	Between servers	Between servers	Between server	Between servers, programs and objects
Itineraries	Special API needed	Special API needed	Special API needed	No special API needed
Persistency	Not possible	Proprietary database only	Not possible	Will be included in VCT 2.1 and VoyagerPro
Scalability	Non	Agents Collaboration	Non	Possible using Space
Multicast Messaging	Non	Yes	Non	Yes
Publish/Subscribe	Non	Non	Non	Yes
Apple connectivity	Restricted	Restricted	Restricted	Full
Security Manager	Yes	Yes	?	Yes

In all the systems an agent must extend the system Agent class, except in Voyager, where an agent must only implement the Agent Interface.

Creating new agents

To create a new agent, you must call a system method (Aglets.createAglet() if you are working with IBM Aglets, or Factory.create() with Voyager). Once created, you get a proxy

to the remote object. This proxy is used to handle all the communication with the agent. You can also obtain new proxies to remote agents later. When the new object is created, a method of the object is called to let the agent initialised correctly.

With Aglets, you can define an `onCreation()` method, and put there the required actions that the agent must do when it is created. On Voyager, the method called is the constructor of the class.

Sending Java messages to remote agents

In the agent, you can have the methods that you desire. The difference is in how to call those methods from another agent.

In an Aglet, you must implement a message handle that will receive all the messages, and call the referred method. This message handle method will take the message, sees the kind of the message, does the operations defined for this message, and sends back a return value. The message is a class that contains the information about the kind of the messages and the arguments for the referred kind of message. When you want to send a message to an aglet, you must create a message object, and then send it to the aglet.

Voyager uses regular Java syntax for calling agent's methods. If your agent defines a method called `sayHello()`, you just write a normal call to this method, as if it was local: `myAgent.sayHello()`. If the method receives an argument, then it is automatically serialised and sent to the remote agent. There is also the possibility of sending only a reference to the argument. Then the object is not serialised, and the remote agent will get a proxy to the object. It is possible to do the same with the returned objects. You can specify to the agent to send back only a proxy to the object, instead of the whole object.

Aglets and Voyager have synchronous, future and one-way messages. Voyager also offers the possibility to send a message to a group of agents. These messages are always asynchronous and do not return any value.

Mobility

Only classes extending the Aglet class can be moved from one host to another. With Voyager, an agent is not different from any other class. You can move any serialisable class at runtime, without any preprocess. Voyager has added a new feature called Facet, with which you can add information to an existing class at runtime, without modifying the code. This is the way voyager allows you to move any serialisable class. You only need to obtain a Mobility facet for the object, and then, you can send it anywhere.

Both systems include some methods to obtain the agents that are running in one host. This can be used to obtain a proxy to these agents, and for security control. Tahiti (the IBM Aglet server) has a visual interface where you can see the messages sent by the agents to the server console, and a trace of the agents that have been in the server.

Voyager does not offer this interface. It also has a server program that can be call from the command line with the same properties than Tahiti, except the visual interface. Voyager

also offers the possibility to start a server inside a program. This can be used to personalise the server behaviour, and for specifying the security restrictions that should be used.

Security

Both systems come with a security system. In voyager, you must specify that you want to install it. It's not the default. You can extend this security class to set up your own security restrictions for the system, and specifically for the agents and host that will interact with the server.

Although IBM's Aglet is a good agent system, actually Voyager offers more facilities for creating agents and working with them. You don't need to learn any special syntax to work with it. Voyager uses the interface to create proxies for remote objects, and the Facets to add new information and behaviour for any class (even without the source code) at runtime.

Both of them offer similar security options. IBM Aglets has its own security system, while Voyager uses the default one that came with the Java Virtual Machine. It also have a security class that can be installed using the default JDK command to install a security manager: `System.setSecurityManager()`.

In the next release of Voyager (VoyagerPro), there will be a new security system that will include:

- Tunnel through popular firewalls via SOCKS protocol
- Authenticate and securely transmit reliable data between Internet clients and servers over the SSL protocol
- Protect against unauthorised use by authenticating and/or verifying credentials of permissions against Access Control List or third party services.

Voyager includes the possibility of accessing to other servers, like CORBA servers, DCOM, or RMI. The Voyager server can also be used as a CORBA server, and receive CORBA calls, or RMI calls.

It is also possible to combine Voyager agents with servlets. I did not test with Aglets, but it should be possible too.

There is a similar thing with applets. A Java applet can create and send agents to any host (it is not restricted to only the web server). There must be a voyager server running in the same host than the web server, and it will be used as router for all the messages. With this addition, an applet can open a database or access to files in the server, with the same restriction than a normal agent.

11 Some Applications of Mobile Agent Technology in HEP Environments

The aim of this paper is not to study all the possibilities that agents may have in HEP environments but to study some of the many applications that this new communication paradigm can offer to the physics community.

RD45 project at CERN has been working to solve the data management problems posed by the LHC experiments, where data volumes of up to 100 PetaBytes and data rates of up to 1.5 GigaBytes/second are expected. RD45 proposes the use of an ODMG compliant Object Database (ODBMS), Objectivity/DB, together with a thin layer of HEP-specific code, plus a coupling to a Mass Storage System, as a solution to the object-persistence problem. This solution for persistence has been adopted by many experiments at CERN, new methods for accessing and configuring the data is being studied, one of them is based on agent technology. This chapter will explain the two applications based on agents that we have developed in 1998.

Before presenting the applications we just introduce very shortly Objectivity/DB for those who do not know it yet. Objectivity/DB is an object oriented and distributed database management system. The highest logical level in the storage hierarchy is the **Federated Database**. This FDB can be divided in several **autonomous partitions**; each of them is formed by one or more data servers (called **AMS**), a **lock server**, and **databases and replicas** of databases which map to files. The objects are contained in the databases and can be accessed directly from an application. One of the objectives of LHC experiments is to distribute data world wide, allowing remote institutes to access data locally without having to access the WAN.

One application of agent technology in such a distributed environment is the management of the federated database. We started building a tool that was for configuration such a complex and large federated database. Later on we decided to introduce on it agents which would control the status of the several database servers and help in an automatic way the database administrator.

Another interesting application of the agent technology is to retrieve information from the ODBMS. Wide area networks have a lower bandwidth than local networks. When you have a distributed database across a WAN, the access to the information can suffer delays due to the network load.

Using a multi-agent system that moves to the remote host containing the requested information avoids the communication across a WAN. The agent can access the information locally, with an important communication improvement. If the agent is sent to the host that contains the database, then no network is required. The agent can analyse the retrieved information before send it back, compute the data and send only the results.

ODBMS Management and Configuration

In this chapter we present briefly a tool for configuration management of the whole Objectivity federated database. For more detailed information of DRO_TOOL see white paper at <http://wwwinfo.cern.ch/asd/cernlib/rd45/whitepapers/9809/DatabaseAdministrationTool.html>.

Agents can be used for many purposes. In the DRO TOOL, agents are in charge of collecting information about the servers involved in an Objectivity/DB federation. The tasks that agents can do are:

- **Status of the AMS and Lock Server:** DRO TOOL agents can watch the status of the lock server and AMS server. If one of the servers stop, the agent can send an email to the specified address, try to restart it, or just ask to the user for an action.
- **Start/Stop an AMS server:** Objectivity/DB tools are only able to start and stop the AMS server locally: agents extend this functionality to allow the user to start the server remotely. An agent can be moved to the specified host, and start or stop there the server.
- **Check the status of the network:** agents located in remote hosts can send messages between them, to test the network bandwidth. They can save the status of the network in the time to display statistics. This can be used to find the best moment to do some operations, like replicas of databases or updates.

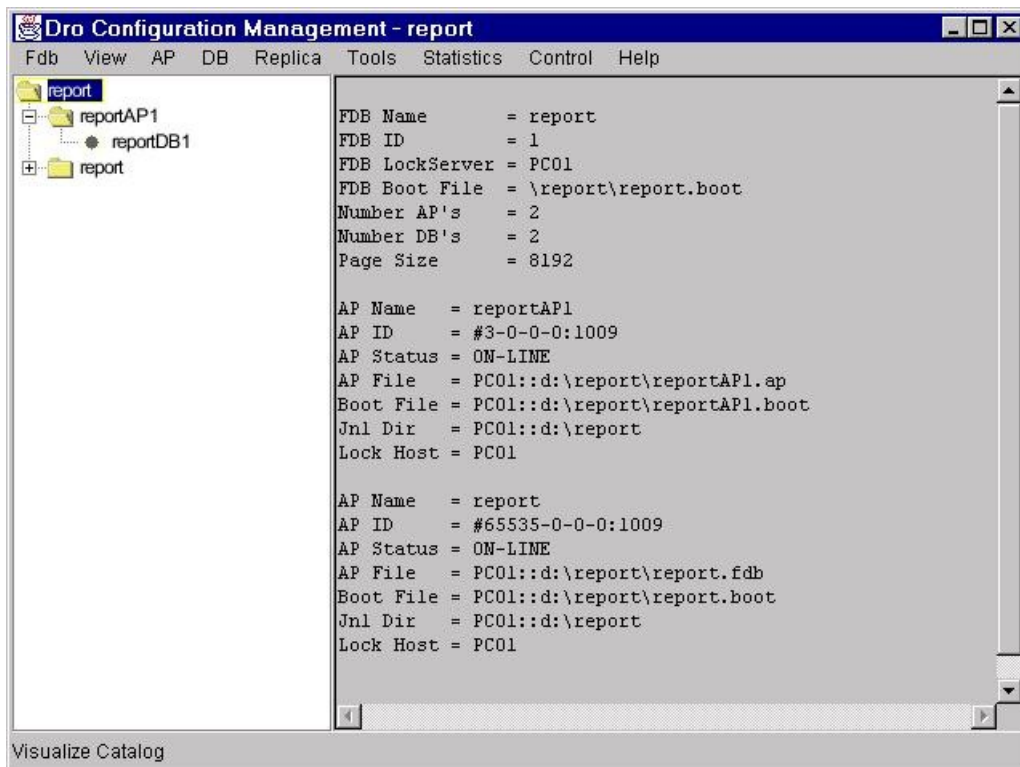


Figure 1 Screen shot of main window DRO_TOOL

Access to remote databases

Another use of agents is to access to an Objectivity/DB from the web but it is not possible to do this directly from an applet,

The other type of agent we have developed is to access to Objectivity/DB federated databases from the web. From a Java Applet, you cannot open a federated database and work with it due to security restrictions in the browser, but using agents it is possible!

The structure of an application using agents would be the following one:

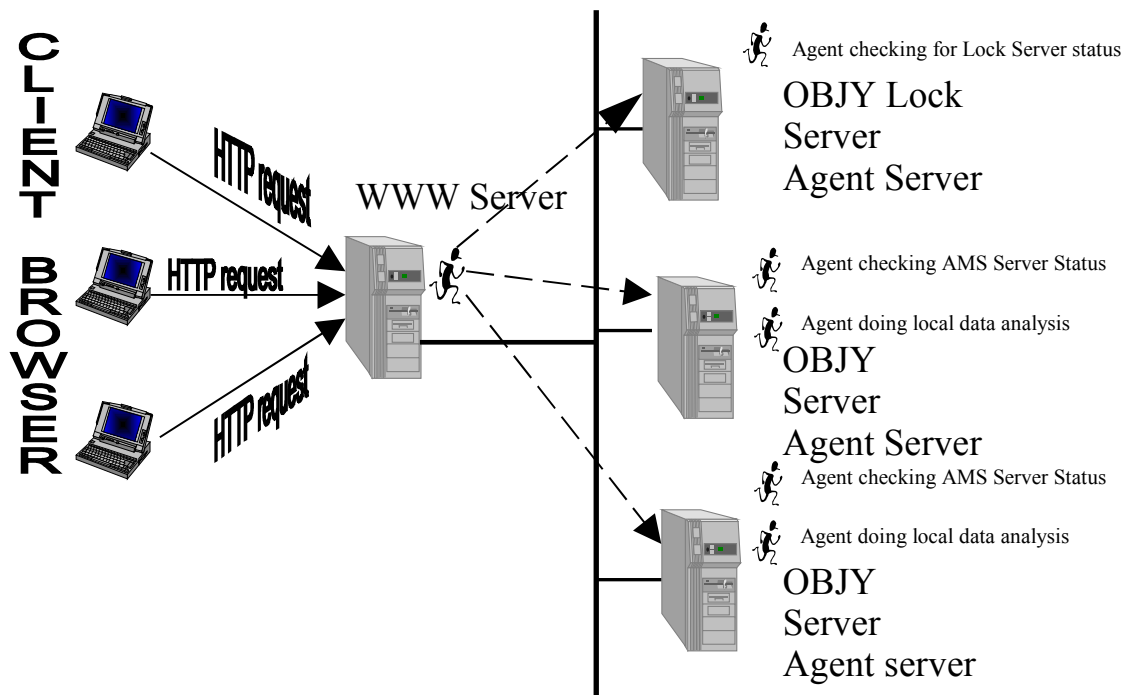


Figure 2 Querying database information from the Web

You could force your application to communicate with the federation only using the AMS server, but even in this case, you will get a security exception in the browser.

In this case, the agent implements all the methods for accessing the database. The Java applet starts a voyager server, and sends an agent to the host where the database resides. An applet can only communicate with the server in which the web page resides. To avoid this limitation, voyager implements a router service. All the messages sent to any agent will pass through this server. With it, you can create a voyager server in the same host as the web server, and use it as a router. Agents created in applets will have no address limitation, and can be sent and communicate with any host without any special restriction.

As the applet need to have access to the Objectivity for Java bindings, the agent must implement all those methods. Objectivity/DB for Java binding must be installed in the host where the agent will be sent (but it is not needed in the host where the applet is running or even in the web server host).

With Objectspace's Voyager 2.0, the Java classes that will be used as agents don't need to specify any special voyager code. Once you create a class, you must create an interface for this class. To do that, you can use the 'igen' program that comes with voyager. It will automatically generate the interface. In the applet, once you have created the agent with the Factory.create() method, you can access to the remote class as if it was local.

The clients need only a Java capable browser, where they execute an applet. In the client side, you don't need any special software to access to Objectivity Databases, or any agent packages. All the classes required by the applet will be loaded from the web server.

The main steps of the application are:

1. Applet bytecodes are loaded from the web server into the browser.
2. The applet starts an agent, and send it to a remote host.
3. Depending of the agent task, it will check for the status of one of the servers or both, or to retrieve the data of a database locally and perform some analysis to this data.

While the agent is doing a task, the client doesn't need to stay connected to the network. The client can connect later to monitor the task, or to retrieve the final result, once the task is done.

Code Examples

In appendix B you will find commented code examples of an agent and an applet which perform this task. In appendix A you will find an example of servlet (servlets where explained in section 9.3). They are based on Objectivity 5.0, Voyager 2.0.0 and jdk1.1.7.

Glossary of terms

- ASDK: Aglets Software Development Kit from IBM.
- ATP: the Agent Transfer Protocol is the one used by IBM's Aglets for the communication and collaboration between agents.
- CGI: Common Gateway Interface is a standard way for a Web server to pass a Web user's request to an application program and to receive data back to forward to the user.
- CORBA: Common Object Requested Broker Architecture is an architecture and specification for creating, distributing, and managing distributed program objects in a network. It allows programs at different locations and developed by different vendors to communicate in a network through an "interface broker".
- DCOM: Distributed Component Object Model is a set of Microsoft concepts and program interfaces in which client program objects can request services from server program objects on other computers in a network.
- Federated database: is a highest logical view of an Objectivity/DB federation. It contains the autonomous partitions and databases.
- IDL: Interface Definition Language is a generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language.
- IIOP: Internet Inter-ORB Protocol is an object-oriented protocol that makes it possible for distributed programs written in different programming languages to communicate over the Internet.
- JDBC: Java Database Connectivity is an API for accessing to a database using the ODBMS.
- KQML: Knowledge Query and Manipulation Language is a language and protocol for exchanging information and knowledge. It is used in Artificial Intelligence.
- MASIF: Mobile Agent System Interoperability Facility; OMG standard for the communication between agents, used by IBM Aglets.
- Objectivity/DB: an object oriented and distributed database.
- OMG: Object Management Group.
- RMI: Remote Method Invocation is the Java equivalence to the RPC protocol.
- RPC: Remote Procedure Call is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details.
- SOCKS: is a protocol that a proxy server can use to accept requests from client users in a company's network so that it can forward them across the Internet.
- SSL: Secure Sockets Layer is a program layer created by Netscape for managing the security of message transmissions in a network.

APPENDIX A: A servlet that retrieves information from the federated database

```
/* ***** */
/*
/* RetrieveInfo.java
/* Shows the information of a federated database
/* @ Javier.Conde@cern.ch, 12/1998
/*
/* ***** */

import java.util.*;
import java.io.*;
// Servlet packages
import javax.servlet.*;
import javax.servlet.http.*;
// OBJY packages
import COM.objy.db.*;
import COM.objy.db.app.*;

public class RetrieveInfo extends HttpServlet
{
    // Session object and FD object (common for all the connection)
    Connection connection = null;
    Session tx;
    ooFDObj fd;
    String bf = null;

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    /** Present the form to fill up with the boot file */
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<HTML><HEAD><TITLE>Display OBJY Federated " +
            "Database information</TITLE></HEAD>");
        out.println("</HEAD><BODY>");
        out.println("<H1>Display OBJY Federated database information</H1>");
        out.println("<HR><FORM METHOD=POST>");
        out.println("Enter the boot file address: <INPUT TYPE=text NAME=bootfile>");
        out.println("<INPUT TYPE=SUBMIT NAME=action VALUE=\"Display\" +
            \" Information\">");
        out.println("</FORM><BR><HR><A HREF=\"mailto:Javier.Conde@cern.ch\">
            + \"Javier Conde</a><BR>");
        out.println(new Date().toString() + "<BR></BODY></HTML>");
        out.close();
    }

    /** Post method that retrieve the bootfile info and use it to open
        the Federated database and shows all the information about the Database */
    protected void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        res.setContentType("text/html");

        // Obtains the bootfile parameter
        String bootfile = req.getParameter("bootfile");
        String msg = "No info";
        // If bootfile is null, return error message
        if(bootfile == null) {
            res.sendError(res.SC_BAD_REQUEST, "No bootfile address specified.");
            return;
        }
        // Retrieve the info from the federated database
        if(req.getParameter("action").equals("Display Information")) {
```

```

        msg = retrieveInfo(bootfile);
    }
    // Creates the output HTML file with the information obtained
    PrintWriter out = res.getWriter();
    out.println("<HTML><HEAD><TITLE>Display information for " + bootfile +
        "</TITLE></HEAD><BODY>");
    out.println(msg);
    out.println("<HR><A HREF=\"mailto:Javier.Conde@cern.ch\">" +
        "Javier Conde</A><BR>\n");
    out.println("Generated at " + new Date().toString() + "<BR>\n");
    out.println("Request from: " + req.getRemoteHost() + " (" +
        req.getRemoteAddr() + ")<BR>\n</BODY></HTML>");
    out.close();
    System.out.println("doPost() finished");
}

/** Present the Servlet info when requested */
public String getServletInfo() {
    return "Retrieve Information of Federated databases, " +
        "by Javier Conde. 11/1998";
}

/** Return the federated database info in HTML format */
private synchronized String retrieveInfo(String bootfile) {
    StringBuffer sb = new StringBuffer();

    // Opens a connection to the federated database
    try {
        if (bf == null) {
            bf = bootfile;
            if ((connection != null) && (connection.isOpen())) {
                connection.close();
            }
            // the connection is opened as read-only
            connection = Connection.open(bootfile, oo.openReadOnly);
        } else {
            connection = Connection.current();
            System.out.println(connection);
            connection.setOpenMode(oo.openReadOnly);
            connection.reopen();
        }
        // Creates a new Session object
        tx = new Session();
        // obtains the federated database object from this connection.
        fd = tx.getFD();

        // begins a new transaction to obtain the Federated database name
        tx.join();
        tx.setOpenMode(oo.openReadOnly);
        tx.begin();
        sb.append("<H2>Information from the FDB <B>" + fd.getName() +
            "</B></H2><BR>");
        tx.commit();
    } catch (DatabaseNotFoundException e1) {
        return new String("<H2>Error opening the federated" +
            " database</H2><br>\nThe federated database cannot be" +
            " found<br>" + e1.getMessage());
    } catch (DatabaseOpenException e) {
        return new String("<H2>Error opening the federated" +
            " database</H2><br>\nOnly one Federated Database can be open" +
            e.getMessage());
    } catch (Exception e2) {
        return new String("<H2>Error opening the federated" +
            " database</H2><br>\n" + e2.getMessage());
    }

    // Call to the local functions to get information about the Fdb, APs and DBs
    sb.append(this.getFdbInfo());
    sb.append(this.getAPInfo());
    sb.append(this.getDBInfo());
    // Once finished, it closes the connection
    tx = null;
}

```

```

        fd = null;
        try {
            connection.close();
            connection = null;
        } catch (Exception e) {
            System.out.println(e.toString());
        }

        // And return the HTML formatted text with the information
        return sb.toString();
    }

    /** Get the properties of the Federated Database */
    public synchronized String getFdbInfo() {
        StringBuffer info = new StringBuffer();
        tx.join();
        tx.setOpenMode(oo.openReadOnly);
        tx.begin();
        try {
            info.append("<B>FDB Name</B>          = " + fd.getName() + "<BR>\n");
            info.append("<B>FDB ID</B>           = " + fd.getNumber() + "<BR>\n");
            info.append("<B>FDB LockServer</B>       = " + fd.getLockServerName() + "<BR>\n");
            info.append("<B>FDB Boot File</B>       = " + connection.getBootFilePath() +
                "<BR>\n");
            info.append("<B>Page Size</B>           = " + fd.getPageSize() + "<BR>\n");
        } catch (Exception e) {
            tx.abort();
            info.append("<H2>Error getting information of the Federated" +
                " Database</H2><BR>\n" + e.getMessage() + "<BR><BR>\n");
            return info.toString();
        }
        tx.commit();
        info.append("<BR>\n");
        return info.toString();
    }

    /** Get the properties of the Autonomous Partitions contained in the current federation */
    public String getAPIInfo() {
        StringBuffer info = new StringBuffer();
        ooAPObj ap;

        tx.join();
        tx.setOpenMode(oo.openReadOnly);
        tx.begin();
        try {
            Iterator apItr = fd.containedAPs();
            while (apItr.hasMoreElements()) {
                info.append("<BR><TABLE BORDER=3>\n");
                ap = (ooAPObj) apItr.nextElement();
                info.append("<TH><B>AP Name</B>: " + ap.getName() + "</TH>\n");
                info.append("<TR><TD>AP ID</TD><TD>" + ap.getOid().getStoreString() +
                    "</TD></TR>\n");
                if (ap.isOnline())
                    info.append("<TR><TD>AP Status</TD><TD>ON-LINE</TD></TR>\n");
                else
                    info.append("<TR><TD>AP Status</TD><TD>OFF-LINE</TD></TR>\n");
                info.append("<TR><TD>AP File</TD><TD>" + ap.getSystemDBFileHost()
                    + " : " + ap.getSystemDBFilePath() + "</TD></TR>\n");
                info.append("<TR><TD>Boot File</TD><TD>" + ap.getBootFileHost() +
                    " : " + ap.getBootFilePath() + "</TD></TR>\n");
                info.append("<TR><TD>Jnl Dir</TD><TD>" + ap.getJournalDirHost() +
                    " : " + ap.getJournalDirPath() + "</TD></TR>\n");
                info.append("<TR><TD>Lock Host</TD><TD>" + ap.getLockServerHost()
                    + "</TD></TR>\n");
                info.append("</TABLE>\n");
            }
        } catch (Exception e) {
            tx.abort();
            info.append("</TABLE>\n<BR><HR><H2>Error getting information of an" +
                " Autonomous Partition</H2><BR>\n" + e.getMessage() +
                "<BR><HR><BR>\n");
        }
    }

```

```

        return info.toString();
    }
    tx.commit();
    return info.toString();
}

/** Get information about the DBs contained in the federation */
public String getDBInfo() {
    StringBuffer info = new StringBuffer();
    ooDBObj db = null;
    ooAPObj ap;

    tx.join();
    tx.setOpenMode(oo.openReadOnly);
    tx.begin();
    try {
        tx.setOfflineMode(oo.IGNORE);
        Iterator dbItr = fd.containedDBs();
        while (dbItr.hasMoreElements()) {
            info.append("<BR><TABLE BORDER=3>\n");
            db = (ooDBObj) dbItr.nextElement();
            db.lock(oo.READ);
            info.append("<TH><B>DB Name</B>: " + db.getName() + "</TH>\n");
            info.append("<TR><TD>DB ID</TD><TD>" + db.getOid().getStoreString() +
                "\n");
            if (!db.isReplicated()) {
                info.append("<TR><TD>DB File</TD><TD>" + db.getHostName() +
                    ":@" + db.getFileName() + "\n");
                info.append("<TR><TD>Contained in</TD><TD>" +
                    db.getContainingPartition().getName() + "</TD></TR>\n");
            } else {
                info.append("<TR><TD>Number of replicas</TD><TD>" +
                    db.getImageCount() + "</TD></TR>\n");
                Iterator itr = db.containingImage();
                while (itr.hasMoreElements()) {
                    ap = (ooAPObj) itr.nextElement();
                    info.append("<TR><TD><B>Contained in " + ap.getName()
                        + "</B></TD></TR>\n");
                    if (db.getTieBreaker() != null) {
                        info.append("<TR><TD>Tie-breaker</TD><TD>" +
                            db.getTieBreaker().getName() + "</TD></TR>\n");
                    }
                    info.append("<TR><TD>DB Image Weight</TD><TD>" +
                        db.getImageWeight(ap) + "</TD></TR>\n");
                    info.append("<TR><TD>DB Image File</TD><TD>" +
                        db.getImageHostName(ap) + ":@" +
                        db.getImageFileName(ap) + "</TD></TR>\n");
                }
            }
            info.append("<TR><TD>Number of containers</TD><TD>" +
                db.getContainerCount() + "</TD></TR>\n");
            Iterator it = db.contains();
            if (it != null) {
                while (it.hasMoreElements()) {
                    info.append("<TR>\n");
                    ooContObj cont = (ooContObj) it.nextElement();
                    info.append("<TR><TD>Container " + cont.getName() +
                        ":@"</TD></TR>\n");
                    info.append("<TR><TD>ID</TD><TD>" + cont.getOid() +
                        "</TD></TR>\n");
                    info.append("<TR><TD>Page count</TD><TD>" +
                        cont.getPageCount() + "</TD></TR>\n");
                    info.append("<TR><TD>Growth factor</TD><TD>" +
                        cont.getGrowthFactor() + "</TD></TR>\n");
                    if (cont.getControlledBy() != null) {
                        info.append("<TR><TD>Controlled by</TD><TD>"
                            + cont.getControlledBy().getName() +
                            "</TD></TR>\n");
                    }
                    info.append("</TR>\n");
                }
            }
        }
    }
}

```

```
        info.append("</TABLE>\n");
    }
    tx.setOfflineMode(oo.ENFORCE);
} catch (Exception e) {
    tx.abort();
    info.append("</TABLE><BR><HR><H2>Error getting information of a" +
        " database</H2><BR>\n" + e.getMessage() + "<HR><BR>");
    return info.toString();
}
tx.commit();
return info.toString();
}
}
```


APPENDIX B: An applet that creates an agent, send it to a remote host, and retrieve the information of a federated database.

```
/* *****/
/*
/* RetrieveInfoApplet.java
/* Shows the information of a federated database
/* @ Javier.Conde@cern.ch, 12/1998
/*
/* *****/

import java.util.*;
import java.io.*;
// Applet package
import java.applet.Applet;
// Voyager packages
import

public class RetrieveInfoApplet extends Applet
{
    TextArea text;
    IOBJYAgent agent;

    public void init() {
        this.setLayout(new BorderLayout());
        text = new TextArea(80, 25);
        this.add("Center", text);
    }

    public void start() {
        try {
            // initialize voyager using the current security sandbox boundaries
            Voyager.startup( this, null );

            // Send agent to the remote host
            agent = (IOBJYAgent) Factory.create("remote.host", "OBJYAgent");

            // Get information from the federated database
            // and display the result in the Text Area
            text.setText(agent.retrieveInfo());

        }
        catch( Exception exception ) {
            System.err.println( exception );
        }
    }

    public void stop() {
        try {
            Voyager.shutdown();
        }
        catch( Exception exception ) {
        }
    }
}

public class OBJYAgent implements Serializable {

    // Session object and FD object (common for all the connection)
    Connection connection = null;
    Session tx;
    ooFDObj fd;
    String bf = null;

    /** Return the federated database information */
    private String retrieveInfo(String bootfile) {
        StringBuffer sb = new StringBuffer();
    }
}
```

```

// Opens a connection to the federated database
try {
    if (bf == null) {
        bf = bootfile;
        if ((connection != null) && (connection.isOpen())) {
            connection.close();
        }
        // the connection is opened as read-only
        connection = Connection.open(bootfile, oo.openReadOnly);
    } else {
        connection = Connection.current();
        System.out.println(connection);
        connection.setOpenMode(oo.openReadOnly);
        connection.reopen();
    }
    // Creates a new Session object
    tx = new Session();
    // obtains the federated database object from this connection.
    fd = tx.getFD();

    // begins a new transaction to obtain the Federated database name
    tx.join();
    tx.setOpenMode(oo.openReadOnly);
    tx.begin();
    sb.append("Information from the FDB " + fd.getName() + "\n");
    tx.commit();
} catch (DatabaseNotFoundException e1) {
    return new String("Error opening the federated database\nThe federated " +
        "database cannot be found\n" + e1.getMessage());
} catch (DatabaseOpenException e) {
    return new String("Error opening the federated database\nOnly one " +
        "Federated Database can be open\n" + e.getMessage());
} catch (Exception e2) {
    return new String("Error opening the federated database\n" + e2.getMessage());
}

// Call to the local functions to get information about the Fdb, APs and DBs
sb.append(this.getFdbInfo());
sb.append(this.getAPIInfo());
sb.append(this.getDBInfo());
// Once finished, it closes the connection
tx = null;
fd = null;
try {
    connection.close();
    connection = null;
} catch (Exception e) {
    System.out.println(e.toString());
}

// And return the HTML formatted text with the information
return sb.toString();
}

/** Get the properties of the Federated Database */
public String getFdbInfo() {
    StringBuffer info = new StringBuffer();
    tx.join();
    tx.setOpenMode(oo.openReadOnly);
    tx.begin();
    try {
        info.append("FDB Name          = " + fd.getName() + "\n");
        info.append("FDB ID            = " + fd.getNumber() + "\n");
        info.append("FDB LockServer    = " + fd.getLockServerName() + "\n");
        info.append("FDB Boot File     = " + connection.getBootFilePath() + "\n");
        info.append("Page Size         = " + fd.getPageSize() + "\n");
    } catch (Exception e) {
        tx.abort();
        info.append("Error getting information of the Federated Database\n" +
            e.getMessage() + "\n");
        return info.toString();
    }
}

```

```

    }
    tx.commit();
    info.append("\n");
    return info.toString();
}

/** Get the properties of the Autonomous Partitions contained in the current federation
 */
public String getAPInfo() {
    StringBuffer info = new StringBuffer();
    ooAPObj ap;

    tx.join();
    tx.setOpenMode(oo.openReadOnly);
    tx.begin();
    try {
        Iterator apItr = fd.containedAPs();
        while (apItr.hasMoreElements()) {
            info.append("\n");
            ap = (ooAPObj) apItr.nextElement();
            info.append("AP Name:" + ap.getName() + "\n");
            info.append("AP ID : " + ap.getOid().getStoreString() + "\n");
            if (ap.isOnline())
                info.append("AP Status: ON-LINE\n");
            else
                info.append("AP Status: OFF-LINE\n");
            info.append("AP File  :" + ap.getSystemDBFileHost() + ":@" +
                ap.getSystemDBFilePath() + "\n");
            info.append("Boot File:" + ap.getBootFileHost() + ":@" +
                ap.getBootFilePath() + "\n");
            info.append("Jnl Dir  :" + ap.getJournalDirHost() + ":@" +
                ap.getJournalDirPath() + "\n");
            info.append("Lock Host:" + ap.getLockServerHost() + "\n");
            info.append("\n");
        }
    } catch (Exception e) {
        tx.abort();
        info.append("\nError getting information of an Autonomous Partition\n" +
            e.getMessage() + "\n");
        return info.toString();
    }
    tx.commit();
    return info.toString();
}

/** Get information about the DBs contained in the federation */
public String getDBInfo() {
    StringBuffer info = new StringBuffer();
    ooDBObj db = null;
    ooAPObj ap;

    tx.join();
    tx.setOpenMode(oo.openReadOnly);
    tx.begin();
    try {
        tx.setOfflineMode(oo.IGNORE);
        Iterator dbItr = fd.containedDBs();
        while (dbItr.hasMoreElements()) {
            info.append("\n");
            db = (ooDBObj) dbItr.nextElement();
            db.lock(oo.READ);
            info.append("DB Name: " + db.getName() + "\n");
            info.append("DB ID : " + db.getOid().getStoreString() + "\n");
            if (!db.isReplicated()) {
                info.append("DB File      : " + db.getHostName() + ":@" +
                    db.getFileName() + "\n");
                info.append("Contained in: " + db.getContainingPartition().getName() +
                    "\n");
            } else {
                info.append("Number of replicas: " + db.getImageCount() + "\n");
                Iterator itr = db.containingImage();
                while (itr.hasMoreElements()) {

```

```

        ap = (ooAPObj) itr.nextElement();
        info.append("Contained in: " + ap.getName() + "\n");
        if (db.getTieBreaker() != null) {
            info.append("Tie-breaker: " + db.getTieBreaker().getName() + "\n");
        }
        info.append("DB Image Weight: " + db.getImageWeight(ap) + "\n");
        info.append("DB Image File : " + db.getImageHostName(ap) + ":" +
            db.getImageFileName(ap) + "\n");
    }
}
info.append("Number of containers: " + db.getContainerCount() + "\n");
Iterator it = db.contains();
if (it != null) {
    while (it.hasMoreElements()) {
        info.append("\n");
        ooContObj cont = (ooContObj) it.nextElement();
        info.append("Container " + cont.getName() + ": \n");
        info.append("\tID : " + cont.getOid() + "\n");
        info.append("\tPage count: " + cont.getPageCount() + "\n");
        info.append("\tGrowth factor: " + cont.getGrowthFactor() + "\n");
        if (cont.getControlledBy() != null) {
            info.append("Controlled by: " + cont.getControlledBy().getName() +
                "\n");
        }
        info.append("\n");
    }
}
info.append("\n");
}
tx.setOfflineMode(oo.ENFORCE);
} catch (Exception e) {
    tx.abort();
    info.append("\nError getting information of a database\n" +
        e.getMessage() + "\n");
    return info.toString();
}
tx.commit();
return info.toString();
}
}

```