Mobile Agents and Java Mobile Agents Toolkits

Damir Horvat[1, 3], Dragana Cvetković[1], Veljko Milutinović[1], Petar Kočović[2] and Vlada Kovačević[2]

[1]Department of Computer Engineering
School of Electrical Engineering
University of Belgrade
P.O. Box 35-54, 11120 Belgrade, Serbia, Yugoslavia

[2]TehnicomNET
Bul. Vojvode Mišića 37
11000 Belgrade, Serbia, Yugoslavia
http://www.tehnicom.net

[3]ComNet
Studentski trg 4
11000 Belgrade, Serbia, Yugoslavia
http://www.comnet.co.yu

E-mail: had@galeb.etf.bg.ac.yu, dana@galeb.etf.bg.ac.yu, vm@eft.bg.ac.yu, ekocovic@tehnicom.net

Abstract

This paper gives an overview of what the mobile agents are, what they should do and how they can be implemented in Java. Why Java? The choice to concentrate on Java is evoked by many existing solutions in Java that handles architectural heterogeneity between communicating machines on the net. It seems to be the best available language for making mobile agents roaming through the Internet for the time being.

1. Introduction

To explain mobile agent technology, it is necessary to start right from the beginning, that is to define what the mobile agents really are, what are their advantages, and to determine what skills and knowledge are needed in order to make them effective.

1.1. What are Mobile Agents?

Briefly, a software agent is a piece of software that performs activities on user's behalf, when given instructions. The more sophisticated it is, the fewer instructions it needs. Mobile agents are specific in their ability to travel from host to host and to perform their tasks at remote locations. They are able to communicate with other agents and systems and move within heterogeneous networks.

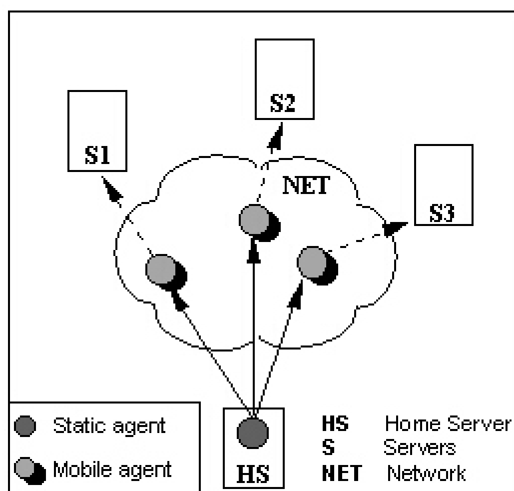

Fig 1: *Static agent sends his mobile agents to their new host severs* (**S1,2,3**)

Proceedings of the HICSS – 2000, Maui, Hawai'i, USA, January 2000.

Additionally, it is crucial for them to be autonomous. Autonomous means that the agent can make its own decisions on how to reach the goals it was given. Instead of the user-initiated interaction via commands, the user is engaged in a cooperative process in which human and computer agents both initiate the communication, monitor the events and perform the tasks [2].

Mobile agent aggregate two things: data (data collected and process states) and code (instructions that direct the behavior). It moves from one host to another, carrying both its data and the code. After arrival, agent continues with execution where it stopped (not from the beginning).

A mobile agent should be able to execute on any machine within a network, regardless of the processor type or operating system. The agent code should not have to be installed on every machine that the agent could potentially visit; it should move with the agent's data automatically [3]. Java Virtual Machine promises it is possible, but under certain conditions. That topic will be discussed later.

1.2. Why Mobile Agents?

Mobile agents are used to implement flexible, scaleable distributed object-oriented systems.

Some people would say: why should one use a mobile agent when the current application would do the same work? Here are some situations where mobile agents are more suitable:

Multi-processor calculations: In the cases of large calculations, often these can be broken into discrete units. Those units can be distributed among a pool of servers or processors for calculations in parallel. Mobile agents would have to take those units to the new host, initiate the calculations, and bring the results back home. Upon completion, all results can be aggregated [4]. One example is the introductory story [see **Fig1**].

Low-reliability or partially-disconnected networks: In systems with low-reliability networks (like notebook computers networked via dial-up connections, for example), it is often hard to fetch large quantities of data, most of which is usually unnecessary [see **Fig2**]. In that situation, mobile agent can be sent to the data source server to do the calculations or to filter the data, according to the given goals, and to bring back only the needed information [see **Fig3**]. Meanwhile the agent's owner does not have to be connected to the net at all [4].

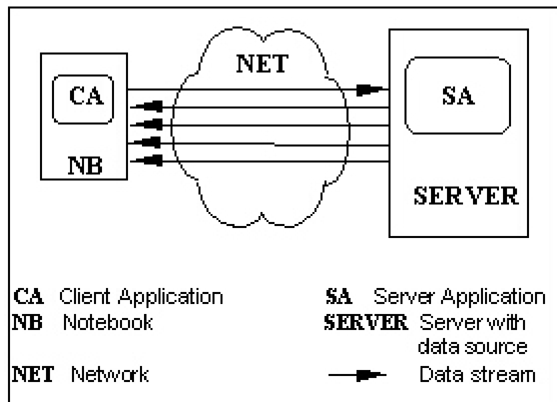

Fig 2: *Current approach: all the data is sent through the network.*

User passivity: For applications that demand immediate reaction to the incoming real-time data, regardless of whether the users are at their desk or not. Example is a sophisticated mail deliverer; besides the e-mail delivery, it can also do some work at the arrival. Difference between this mobile agent and the static one which could do the same work is that the mobile one gets the instructions from the sender and the static one must be prepared for such a work by the receiver.

1.3. Additional Requirements

In the defining of mobile agents, *autonomy* and *mobility* are emphasized as a cornerstone of the agents. Without mobility, they wouldn't be mobile agents but static ones; without autonomy, they wouldn't be agents at all, but directly manipulated applications. These two requirements would be enough if agents spent their lives in some laboratory conditions, but it isn't so. Their purpose is to roam through unknown parts of the Web in search for new data. To complete their mission, additional skills and knowledge are needed. Some of the additional requirements for mobile agents are:

Programmability: Agent must be programmed or instructed in some manner. Programming should be easy and user-friendly as much as possible. Agents should be easily manipulated and created even by persons who are not familiar with programming languages.

Safety: Remote host must ensure that the agent will not commit illegal acts of any kind. Mobile agents are viruses-like programs and without maximum precaution they can do much harm to the hosting system.

Privacy: The agent's internal state and program should not be visible to others. Mobile agents roam through Internet and bring important data (like credit card numbers and so) which could be misused by malicious hosts or by competing companies.

Navigationability: The agent must be able to find the needed resource, on the present host or by traveling from host to host, fulfilling its goals. That is a part of its autonomy.

Communicationability: The agent must be able to communicate not only with the master agent at the host but with other agents, too. Through this communication, an agent can collaborate with other agents in the intention to reach its goals.

Learning: The agent learns about its environment and actions to be more effective. It can take a number of different forms: *deliberative learning* which needs large amount of storage for knowledge and *reflexive learning* that allows for completely "automatic" learning, where initially there is minimal knowledge storage, and there is an incremental increase in storage as the agent learns from its environment. Through communication and collaboration among other agents, some kind of semi-intelligence can be achieved.

Robustness: The agent must be prepared for unexpected situations, which may occur on the net. Not everything can be foreseen but agent must be prepared for breakdowns of connections, hosts or even that it can be destroyed. In such cases, there should be some point where the agent could go back [7]. A checkpoint-restore mechanism can be used to restart agents. The agent's state information is checkpointed before and after execution on a particular server. When a server is restarted, a recovery process is executed which restarts any agents left on the server at last shutdown.

To accomplish their tasks and for security reasons, mobile agents have to carry data about themselves and about their goals. Some of these data are:

Owner: Parent process name or master agent name. Agents can have many owners.

Author: In the case it is needed to contact the author.

Lifetime: Time to live (TTL). Every mobile agent has to have a limited lifetime after which it is terminated in the cases when the agent is out of control or in deadlock.

Account: Agent could be using some resource that has to be paid or buy things which are needed, so there is a billing related information, or a link to owner accounts.

Goal: Measure of success. Every agent must be goal-oriented and must perform its task until the goal is reached.

Subject: Description of the goal's attributes.

Background: Supporting information [7].

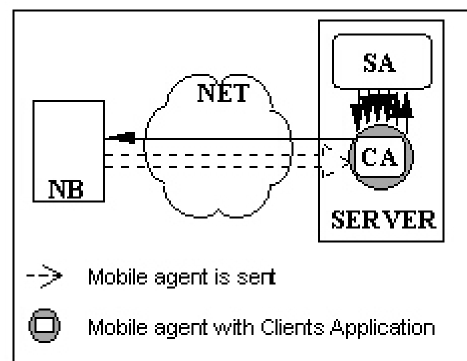

Fig 3: *New approach: the agent is sent to the data source and only important data is sent through the network*

2. Details of Implementations in Java

Maybe one of the most important issues about the mobile agents is the selection of implementation language. In earlier days there were attempts to improve distributed programming and to enable the mobile agent programming which resulted in languages like TCL, Scheme, Oblique, and Rosette; even C and C++ languages were used for this purpose. The only widely accepted language was Telescript by Object Magic but neither it did really ‘took off’. Some efforts were done in using these languages in mobile agent projects like Ara (agents using TCL and C/C++) [14] and D’Agent (earlier called Agent TCL -- using TCL) but they have not gained wider popularity or they are not completed yet.

The reasons for using Java are numerous. Java Developing Kit 1.1 (JDK 1.1) and JDK 1.2 with their possibilities, like Remote Method Invocation (RMI) that allows object methods to be called over the network and the serialization of objects that allows objects to be sent via byte streams for network transmission, are almost a natural choice for implementation of mobility. Source code for software can be transparently downloaded from anywhere on the net. Further more, JDK 1.2 gives some improvements in security which enables fine graduation of the security allowances.

Java is easy to implement on almost any system and thanks to its popularity, there are many platforms deployed already with many services, which the agents can use. Java resembles C++ in many ways, and that makes it an easy-to-use tool for development and debugging for the Internet applications.

2.1. An Agent Server

In theory, agent code should not be installed on every machine that an agent could potentially visit but it is not possible in practice. Mainly because of the security reasons (computer viruses are similar to mobile agents), a mobile agent server (environment) is needed.

The agent server is built on the top of a host system to protect its resources. The server is a sandbox into which the mobile agents move and it provides all services to the agents but limits their actions. The limits are on the amount of resources it can use, the agent's lifetime, the number of possible access to the host or through allowances. New JDK 1.2 provides tools for making these sandboxes more flexible and easier to use [8][15][16].

On the other hand, host has almost complete control over the visiting agents. The agent cannot checkout the host and determines if the interpreter is correct, will the server start the agent correctly or will it transport the agent where it wants.

In some cases the Java classes on the host can be modified and it would modify agents. Solutions are to move only to the trusted hosts or there must be a third side, e.g., the agent’s home system that would detect if something happened to the agent (mechanism of digital signature and encryption) and react on it. Also, the agent server can read all the data that the agent carries (e.g., credit card number) and misuse it for its purpose but this is still the unsolved problem [14].

Other very important role of the server is to enable the agent transfer. To transfer an agent, the local server negotiates with other servers, freezes agent execution, transfers the agent to a remote server and when the agent is received, allows it to resume execution at the remote location. Also, it allows monitoring of all transfers and events in the system by the administrator [8][23].

The agent server has to allow multiple agents to co-exist and to execute on the same server without interfering with each other [8]. The server provides agents with a view of each other and allows communication, but without direct interfering with other agents. If agent could directly invoke the public method of other agents, it would be able to change other agent's data or even destroy it. In cases when an agent needs service from the other agents it has to send a message with the request. The agent that gets the message then can check the request and do the service if it is possible [8][15].

2.2. An Agent

The most important part of every mobile agent system is an agent, itself. The mobile agent migrates between servers and completes its tasks. To do so, the agent must plan the best course of action and if it is stuck, to become aware of it and either to make additional actions or to move to a new environment where it might find the information for progress [13]. This chapter is an inside look at agents in the Java environment.

2.2.1. The Lifecycle Model

The life of a mobile agent is modeled with the stages it goes through called *lifecycle model* [see **Fig 4**]. The stages of the model are:

Creation of the agent is done only once when new agent is created. Every agent gets its unique id, initial state and then it is prepared for further instructions.

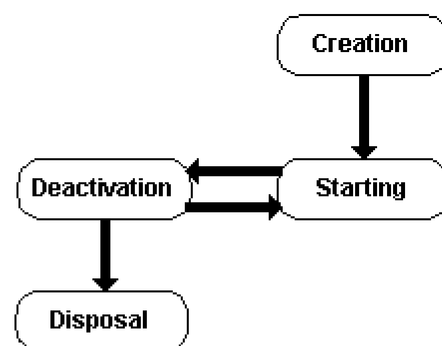

Fig 4: *Life cycle of an agent*

Starting is done each time when the agent arrives to a new host. The agent has its own thread of execution and can execute asynchronously. The server initializes the agent and gives it a thread of execution after which the agent resumes its execution. All the agents are executed in parallel on the host.

After *deactivation*, the agent stops all its calculations and stores its state and intermediate results to a disk. That means, the agent is put to sleep using object serialization, available in JDK1.1 and later. The states of the agent objects are exported to a byte stream and later, they are reconstructed from the byte stream [5].

The deactivation method can also be used for making checkpoints before performing some unsecured operations or moving to unknown host. The possible difficulty is when the agent is recreated from its checkpoint while it is still active on the remote host. To prevent confusion or errors, before using the checkpoint for creation, it must be absolutely certain the original agent is deactivated [14].

Disposal means the agent terminates all its activity and frees all resources it's using. After that, its state is lost forever.

Cloning is process of making multiple clones of an agent using object serialization. New (cloned) agent is identically same with the original except the id number, if any and it is sent to the new host. Cloning is used when more than one agent is needed for completing the task.

2.2.2. Mobility

The most important issue of the mobile agents is their mobility. There are two basic models of migration: the *weak* and the *strong* migration. The weak migration is transfer of only the agent's code and data. The agent restarts on the new host from the beginning but with its data. The agent must prepare for the transfer so that all the necessary information is in the data. The strong migration transfers agent's state, too and the agent restarts from the point where it stopped. The weak migration is commonly used in agent systems today, since strong one can be difficult to implement into the Java environment or costly in performance (greater transfers are needed).

The agents use two mechanisms of migration between the hosts: RMI (widely used) or through *sockets*.

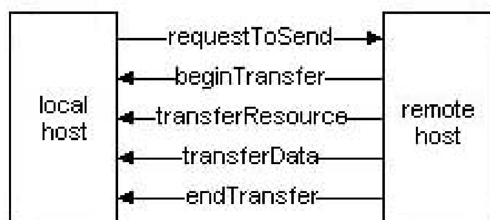

Fig 5: *Agent migration using RMI* [9]. *The Agent Servers initial transfers by invoking public methods*

RMI is a feature of JDK 1.1 where a process can invoke Java public method of remote process. An agent, using RMI migration, first sends a message to his local host demanding transfer to the new host. The local host connects with the requested new host and initializes the transfer invoking public method on the remote host [see **Fig 5**]. From this point, the remote host is responsible for directing the transfer. First, the remote host invokes *BeginTransfer* method on the local host. The local host serializes the agent and prepares it for transfer. The next step is transfer of resource and data of the agent, using RMI for initializing. Finally, the remote host informs the local host that the transfer is completed and it restarts the agent on the new location [9].

In the migration mechanism that uses sockets, the idea is to convert the agent data and code to byte array that would be protocol independent. To do this, the agent invokes public method on the local server after which it is serialized and prepared for transfer by passing it through multiple layers. When the agent is prepared, it can be transferred to the new location by using standard transport protocols (e.g., TCP/IP) [see **Fig 6**].

2.2.3. Communication

The computer science has produced many communication mechanisms that are used in mobile agent's systems. The most used ones are:

Procedure call (synchronous) mechanism: An entity A calls entity B to do a service during which the entity A is blocked. It enables procedure nesting. It is fast and easy to follow through program's flow of execution. The disadvantages are that it is synchronous and difficult to parallelise. RMI is such mechanism.

The callback (asynchronous) mechanism: An entity A calls an entity B to do it a service but continues on with its task. The entity B, when is finished, calls back and gives the result to the entity A. It permits truly asynchronous processing but it is complicated and difficult to follow [10].

The mailbox mechanism: Somewhere between previous two mechanisms. An entity A calls entity B for a service and tells it to put the results into mailbox.

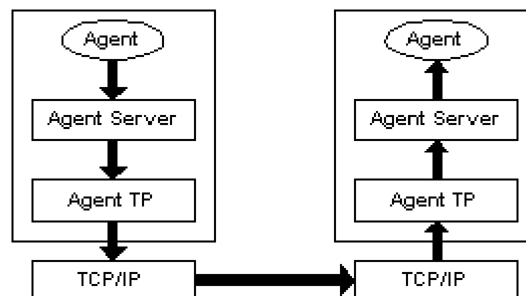

Fig 6: *Migration through sockets. The agent code transferred through multiple layers is transport protocol independent*

The entity A continues on its task, periodically checking its mailbox if the results are there. This is more difficult to implement but it's asynchronous and easy to follow [10].

The most of the mobile agent systems use more than one of these mechanisms as well as broadcast or multicast mechanism for sending a same message to multiple receivers.

3. The Existing Tools

The Java VM, JDK 1.1 as well JDK1.2 with its features make the creating of the mobile agent a fairly simple task. In building of such a system the problem is not to make it work, but to make it safe for its environment and compatible with other agent systems.

There are several Java-based mobile agent systems commercially available for those who want to do some serious work. These mobile agent toolkits provide all needed classes in Java for building such systems. The builder supplies the agents with the "brain", the algorithms that will be used to accomplish the given goals.

Here is an overview of four agent systems that use Java: *IBM's Aglets, Object Space's Voyager, General Magic's Odyssey*, and *MEITCA's Concordia*. These Java-based mobile agent systems seem to be relatively mature and ready for real use. They share certain characteristics: they provide an agent server, the agents can migrate from a server to a server (carrying their state with them), the agents can load their code from variety of sources (filesystems, Web and ftp servers) and they are pure Java, using JDK 1.1. and later [11].

3.1. Aglets

Aglets Software Development Kit (ASDK) is a product of IBM's Tokyo Research Laboratory, initiated in early 1995. The goal has been to bring the flavour of mobility to the applets (Aglet means **ag**ent plus app**let**) and to build a network of Aglets hosts with the task-specific Aglets [18].

The Aglets SDK includes Aglets API, documentation, sample Aglets, the Aglets Server called *Tahiti* and the Agent Web Launcher named *Fiji* [11].

Tahiti is an application program that runs as an agent server. Tahiti has an easy to use GUI and provides a user interface for monitoring, creating, dispatching, and disposing of agents as well as enables setting the agent's access privileges to the agent servers. On a single computer multiple servers can be run by assigning them different port numbers [18].

Fiji is a Java applet capable of creating Aglets or retracting an existing Aglet into the client's Web browser. The Fiji applet takes an agent's URL as its parameter and can easily be embedded in a Web page by using HTML, like any Java applet.

The Aglets architecture consists of the Aglet API (set of Java classes and interfaces that allows creation of mobile agents), the *Aglets Runtime Layer* and two implementation layers, the *Agent Transport and Communication Interface* (ATCI) and the *Agent Transfer Protocol* (ATP) [18].

The Aglets Runtime Layer is an implementation of the Aglets API that provides the fundamental functionality for Aglets to be created, managed and dispatched to remote hosts. Together with ATP and ATCI, it allows Aglets mobility [18].

The Agent Transfer Protocol is an application-level standard protocol for the distributed agent-based information systems. The ATP offers a uniform and platform-independent protocol for transferring the agents between networked computers, using Universal Resource Locators (URL) for the agent resource location [18].

The ATCI is a higher communication level, an independent agent protocol that enables agents to move and communicate within a network. It is a programming interface that enables programmers to develop platform independent agents without building into protocols for wire communication [18].

Aglet system uses migration through sockets mechanism. During a migration, an Aglet is sending a request to the Aglets Runtime Layer. The layer converts the Aglet by serialization into the form of byte array consisting of its data and code. The resulting byte stream is passed to the ATP through the ATCI that makes it protocol-independent (important for heterogeneous networks). The ATP constructs a bit array containing general information about the Aglet system and Aglet's id together with information given from the Aglets Runtime, after which the aglet is ready for transfer. On the remote server the process is the same only inverted [12].

The Aglets use weak migration. Further more, Aglet system does not transfer system classes, it assumes that all the system classes are available at the destination. That reduces necessary transfer but has an impact on security and compatibility: Aglets can approach only systems that are running the Aglets Server [12].

The ASDK supports interesting mechanism for preventing malicious agents to mess with other agents: if deactivation (or such) method is called by other agent, agent will first check should it obey, and can "just say no" if it is not correct.

Aglets support synchronous and asynchronous communication. The communication is supported only locally and remote messages are sent by the *Message Aglets*. The Message Aglets is sent like an ordinary Aglets but no bytecode is transferred, only the message. The Aglets class also provides a subscribe model for messaging where an Aglet can register its interest in particular message types and ignoring others, optimizing message delivery and agent's code [11] [12].

The Aglet system is the most widely used mobile agent system now, thanks to brand name of IBM and many press coverage. From the users point of view the ASDK provides easy-to-install package and easy-to-use GUI that is important as well. The disadvantages are that it does not have a method for saving Aglets state for reason of persistence and security and the problem of compatibility with other systems because it can use only the Aglets classes and servers. One more important problem is that there is no method to send messages to the Aglets while they are moving [11] [12].

The latest versions of Aglet system are ASDK Version 1.0.3 and ASDK 1.1 Beta and they are ready for download at the IBM site. ASDK 1.0 does not have expiring date and it is free for use but ASDK 1.1 Beta will not start if the expire date has been reached.

3.2. Voyager

Voyager is a concept of mobile objects by ObjectSpace, started on mid-1996. It has a unique concept that all serializable objects (Java source code or class file) can be mobile by using *Virtual Code Compiler* (*vcc*). The *vcc* utility reads a .class or .java file and generate a new remote-enabled "virtual class". The new virtual class contains a superset of the original class functions and allows function calls and message passing even when objects are remote or moving. Voyager allows an object to communicate with an instance of a remote-enabled class via a special kind of object called a virtual reference. When messages are sent to a virtual reference, the virtual reference forwards the messages to the instance of the remote-enabled class. If a message has a return value, the target object sends the return value to the virtual reference, which returns it to the sender.

After generating a virtual class, you can use its constructors to create a remote instance of the original class. The virtual reference resides in your current program and references the remote instance. The remote instance may reside in the current or a different program.

When a voyager-enabled program starts, it automatically spawns threads that provide timing services, perform distributed garbage collection and accept network traffic. Every Voyager-enabled program has a network address consisting of its host name and a communications port number, which is an integer unique to the host.

This system provides a good tool for making distributed systems as well as mobile agent systems.

Agents have all the same features as simple objects-they can be assigned aliases, have virtual references, communicate with remote objects, and so on. The main difference between an agent and a moving object is that the agent can move itself autonomously. When an agent moves to another host, it calls it *moveTo( )* method which uses RMI for transfer with a destination address and the name of a callback function.

On the new host, the agent receives the name of the callback function as a message it sent and resumes its execution. Voyager is using weak migration mechanism, too [12].

When an agent is preparing for move, it automatically leaves behind a forwarder with its new location, to forward messages. The messages are delivered by the lightweight messenger agents: when an agent is addressed, the messenger is following the forwarder to the agent. Once agent is found on the new location, that location is recorded as a new starting address for messaging [12].

The Voyager system supports four types of messages: synchronous, one-way (asynchronous), future (mailboxing), one-way multicast and selective multicast. For multicast the Voyager system uses structure called Space™. Space is built of small subspaces connected together and users connect to this virtual Space as it is a single object. When the messages are sent into the Space, they are cloned and then multicast in parallel to all the objects in the Space or selectively [12][19].

The agent persistence is provided by explicit method *saveNow( )* which saves a copy of the agent to the Voyager database. When an agent moves to a remote host, the persistent copy moves with it to the remote database. The local database then contains a forwarder to the new database, similar to the agent's forwarder [12].

The Voyager system has a server called "voyager" but it is not necessary to run such server on all the nodes in network, where virtual objects can migrate. That solution is appropriate to the encapsulated network but otherwise it is potential security threat. For that reason, the Voyager agents have restricted operations they can perform on the host [11][12].

The Voyager system has five different life span schemes for the agents and other virtual objects and the default value is to live until there are no more references to it. The Voyager agents can be set to live for a specified amount of time or until an explicit point of time or until it becomes inactive for specified amount of time as well as it can live forever [12].

Important thing for developers using the Voyager package is a great amount of documentation available. It includes The User Guide and the full API documentation for the package and several dozen examples that should introduce new users to the Voyager. The other advantages of this system are that all serializable objects are moveable which gives a new dimension to the programming as well as that the agents can receive messages while moving [11].

Voyager Core Technology Version 2 is free for most commercial uses and can be downloaded from the Object Space's site. Object Space also offers Voyager Partner Program with business, educational and technical support for system integrators and consulting and product businesses.

3.3. Odyssey

General Magic product Telescript, language for creating mobile agents was the first such tool that actually worked but it was not widely adopted. The Telescript language supplements systems programming languages such as C and C++. Entire applications can be written in the Telescript language, but the typical application is written partly in C. The C parts include the stationary software in user computers that lets agents interact with users, and the stationary software in servers that lets places interact, for example, with databases. The agents and the "surfaces" of places to which they are exposed are written in the Telescript language.

Using their experience in the mobile agent technology, General Magic began to develop Java mobile agent tool called Odyssey. The Odyssey system provides a set of Java class libraries for developing distributed mobile applications [11].

The Odyssey system includes agents, agent system and places. The agent system is a platform that can create, manage, interpret, execute, transfer, and terminate agents. It consists of a set of Java classes to support Odyssey agents and Odyssey places. The agent system has the authority of the region or organization that it represents [20].

Telescript technology models a network of computers as a collection of places. The place is a context within an agent system in which agents execute, similar to the definition of server in chapter 2.2.1. On one host can be more than one place and it is an interface between agent and host's system resource [20].

The Agents are created by subclassing the Odyssey *Agent class* and each has its own thread of execution so that it can perform tasks on its own initiative. In the Odyssey system are two kinds of agents: 'real' agents and workers. A worker is structured as a set of tasks each to be completed at the specific hosts. At each destination, the worker completes the next task on his list and then it moves to the new location. The Odyssey worker may manipulate its task list at any point during its travels, adding new destinations. The Odyssey agent ('real agent') has more independence in completing its tasks, it can move during its execution and it is not bound to the system where it was created [20].

The Odyssey class hierarchy 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 whom an agent wants to communicate with), and *ProcessName* (used to generate the unique names of all processes, including agents and places). It also includes three interfaces: *AgentSystem*, *Finder*, and *Transport* that allow a developer to customize the implementation of an Odyssey agent system [20].

Odyssey supports weak migration mechanism using RMI, similar to the other agent systems. It has no effect on the workers because their tasks are completed on the local host before it moves. The 'real' agents restart on each new location and it has to resume its execution from its data.

This system has a few disadvantages that should be considered in new versions. As the biggest problem for developing mobile agent system is that Odyssey does not have any security mechanism, except which are provided by Java. Also, there is lack of persistence mechanisms for case of system failure or lost of data. The Odyssey system is focused on workers, which is more centralized system than other mobile agents and does not have any communication mechanism.

Odyssey is provided free of charge for research and development (non commercial) purposes and it has not yet been determined will Odyssey be available for commercial use in the future [20].

3.4. Concordia

Concordia by Mitsubishi Electric ITCA (MEITCA) is the youngest mobile agent system but it offers some solutions that makes it a good choice for enterprise applications on the Web.

Concordia is a framework for development and managing network-efficient mobile agent applications for accessing information anytime, anywhere and on any device [21].

Concordia includes a collaboration framework that enables multiple agents to work together and coordinate their actions. Agents within an application may form one or more collaboration units.

In Concordia there is a distributed events framework that enables agents to communicate with each other either synchronously or asynchronously. They are extremely useful for notifying objects of changes to resources and unexpected conditions.

Concordia has the *Concordia server* that includes several modular components that work in concert to provide an integrated development environment and management tool for its agents. These components and their services are as follows:

The *Agent Manager* serves as a communication server for transferring an agent. Also, it manages creation and destruction of the agent. It provides an environment for agent's execution.

The *Administration Manager* provides a GUI for the administration of the Concordia network including all of its services. It permits remote administration of Concordia services running on other nodes, so only one administration manager is required for the entire Concordia network.

The *Security Manager* is responsible for identifying users, authenticating agents, protecting server resources, and authorizing the use of dynamically loaded classes. The security level can be adjusted: from the weak identity check to the strong authentication and security provided from external authorities. The Security Manager’s user interface is integrated into the Administration Manager [21].

The *Persistence Manager* supports the persistence and recovery of the agents after system or network failure. Using the Java serialization methods, the persistent store manager writes the state information of the agent to disk and it may return to this checkpoint if necessary. It’s important that agent itself can request a checkpoint before performing critical procedures.

The *Event Manager* accepts event registrations, listens for and receives events. An agent registers with the Event Manager indicating which events it is interested in to be notified about. The event notification can be sent to the agents on any node in the Concordia network. The Event Manager handles Concordia agent collaboration [12][21].

The *Queue Manager* schedules and reschedules the transport of the agent across the network. The Queue Manager communicates with the local Concordia server and handshakes with the remote Queue Managers for reliable transmissions. The Queue Managers communicate using Java RMI. If the remote system is disconnected from the network, the agent transmission is rescheduled. Coupled with the service of persistence manager, it enables reliability to the Concordia agents.

The *Directory Manager* provides naming service. It may consult a local name server or may be set to pass requests to other existing name servers.

The *Service Bridge* provides the interface from Concordia agents to the services available at the various nodes in the Concordia network. It provides access to the native API as well as interfacing these to the Directory Manager and Service Manager [21].

The *Agent Tools Library* provides all the classes required to build Concordia mobile agents including the Agent class itself [21].

The travel plan of an agent is described by the *Itinerary Set* when an agent is launched. The Itinerary is a separate data structure from the agent, to simplify the agent model and to enable more predictability in where the agent will travel [12].

During the transfer between two hosts Concordia transports agent's code, data, and state information using Java RMI through the Concordia servers and using its Itinerary to determine the next destination. When the agent again begins executing, it is restarted on the new node according to the method specified in its Itinerary. Its security credentials are transferred with it automatically and its access to services is under local administrative control at all times.

There are two types of inter-agent communication in the Concordia system: the distributed asynchronous events and the collaboration.

The distributed asynchronous events have two forms: the selected events and the group-oriented events. An agent receives selected events after it registers with the event manager by sending a list of event types it wants to receive. The group-oriented events offer non-filtered communication between a group of agents after agent register with in event group. The collaboration is used for complex agent co-ordination. This communication method seems specially suited to divide and conquer type problems [see Multi-process calculations, chapter 2.2].

The Concordia system provides modularity, security, and reliable agent transmission and enables agent collaboration. Further more, it has easy-to-use GUI for administration and enables remote administration. The disadvantage is that there is no possibility of direct agent to agent communication.

On the MEITCA’s Web site, a free 30-day evaluation kit (without Security Manager) of the Concordia system v. 1.1 is available. Concordia is basically a commercial product that needs a license agreement. Concordia Partners Program is also offered for system integrators, independent software vendors, value added resellers and information technology departments that provide distributed software solutions to their clients which provides access to MEITCA’s Concordia technology and resources.

4. Conclusion

All the previous mobile agent systems share some characteristics but also have their particularities:

- the Aglet system is nicely accommodated to the Internet environment; it is robust and it is the most widely used system.
- Voyager provides unique concept of serialization and mobility of objects and allows quick and easy creating of sophisticated network applications.
- Odyssey is more distributed systems oriented than other mobile agent systems but brings a new dimension to the programming.
- the Concordia system provides modularity, security and enables remote administration that makes it suitable for enterprise systems.

Each system also has some disadvantages that should be solved in new versions of the systems. One is the security issue; not all the security precautions are implemented in today’s agent systems. Problem is a protection against an attack of a malicious host towards an agent and its data.

The biggest strength of mobile agents is their potential to communicate and collaborate. Through communication they share their experience and information and accomplish their goals easier and faster.

Again, there is a problem of the language they will use. Every agent system provides its own solutions for communication as well as the interface between an agent and its host. It makes problems with compatibility with other systems. One of the solutions for communication is the use of *Knowledge Query and Manipulation Language* (KQML). The complexity of KQML makes it difficult to implement into the agent system. Several companies (including IBM and General Magic) work together on Mobile Agent Facility in Object Management Group which should help to ensure that different agent systems will be able to work together [11].

All the systems today require a knowledge of programming in general that is above the level of an average user. The low-level details (e.g., the Java code) will have to be hidden to provide a very high level abstraction in order for ordinary people to be able to create agents. The beauty of agents is that they can help to solve problems more naturally and simpler than it is traditionally done by distributed computing [12].

Finally, there is an issue of intelligence. That is not directly connected with the mobile agents, they do not really have to be smart to do the job. Intelligence is a question of how good are the algorithms they are supplied. Using their capabilities to work collectively, like in an ant colony, and learning from each other, some kind of semi-intelligence could be achieved**.**

The mobile agent technology is still a new one, existing tools are still under development but this technology will have its share in the WWWorld of tomorrow.

5. References

[1] Mirkovic, J., Kraus, L., Milutinovic, V., "A Survey of Genetic Algorithms for Intelligent Internet Search, " University of Belgrade, Belgrade, Serbia, Yugoslavia, 1998. *http://www.galeb.etf.bg.ac.yu/~sunshine/.*

[2] Maes, P., "Agents that Reduce Work and Information Overload, "*Communication of the ACM*, Vol. 37, No. 7, July 1994.

[3] Farley, S. R., *Mobile Agent System Architecture,* SIGS Publications, New York, New York, USA, 1997.

[4] Sommers, B., "Agents: Not just for Bonds anymore, " *Javaworld,* April 1997. http://www.javaworld.com/ */jw-04-1997/jw-04-agents.html*

[5] Venners, B., "Under the Hood: The architecture of aglets, " *Javaworld,* April 1997. *http://www.javaworld.com/jw-04-1997/jw-04-hood.html*

[6] Venners, B., "Solve real problems with aglets, a type of mobile agent, " *Javaworld*, May 1997. *http://www.javaworld.com/jw-05-1997/jw-05-hood.html*

[7] Kalakota, R. Whinston, A., *Frontier of Electronic Commerce,* Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1996.

[8] Sundsted, T., "An Introduction to agents," *Javaworld,* June 1998. *http://www.javaworld.com/jw-06-1998/ /jw-06-howto.html*

[9] Sundsted, T., "Agents on the move, " *Javaworld,* July 1998. *http://www.javaworld.com/jw-07-1998/ /jw-07-howto.html*

[10] Sundsted, T., "Agents Talking to Agents, " *Javaworld,* September 1998. *http://www.javaworld.com/jw-09-1998/ /jw-09-howto.html*

[11] Kiniry, J., Zimmerman, D., "A Hands-On Look at Java Mobile Agents, " *IEEE Internet Computing*, Volume I, Number 4, July/August 1997.

[12] Shah, K., Guota, R., Timm, S., "Study of Mobile Agent Systems, " Department of Computer Science, Virginia Tech, Blachsburg, Virginia, USA, 1998. *http://csgrad.cs.vt.edu/~stimm/agents/*

[13] Ohsuga, A., Nagai, Y., Irie, Y., Hattori, M., Honiden, S., "Plangent: An Aproach to Making Mobile Agents Intelligent, " *IEEE Internet Computing*, Volume I, Number 4, July/August 1997.

[14] Peine, H., Stolpmann, T., "The Architecture of the Ara Platform for Mobile Agents, " Department of Computer Science, University of Kaiserslautern, Germany, 1998. *http://www.uni-kl.de/AGNehmer/Projekte/Ara/Doc/ /architecture.ps.gz*

[15] Karjoth, G., Lange, D. B., Oshima, M., "A Security Model For Aglets, " *IEEE Internet Computing*, Volume I, Number 4, July/August 1997.

[16] Kotz, D., Gray, R., Nog, S., Rus, D., Chawla, S., Cybenko, G., "Agent TCL:Targeting the Needs of Mobile Computers, " *IEEE Internet Computing*, Volume I, Number 4, July/August 1997.

[17] Pertie, C. J., "What's An Agent… And What's So Intelligent About It?, " *IEEE Internet Computing,* Volume I, Number 4, July/August 1997.

[18] *IBM Aglets Workbench, A White Paper,* IBM Tokyo Research Laboratory, Tokyo, Japan, 1996. http://www.trl.ibm.co.jp/aglets/whitepaper.html

[19] *ObjectSpace Voyager Core Package Technical Overview,* ObjectSpace, Dallas, Texas, USA, December 1997. http://www.objectspace.com/voyager/whiteparer/ */RMIComparationW97.PDF*

[20] *Introduction to The Odyssey API,* General Magic, Sunnyvale, California, USA, 1997. *http://www.genmagic.com/agents/odysseyIntro.ps*

[21] *Concordia: An Infrastructure for Collaboration Mobile Agents,* Mitsubishi Electric ITA, Waltham, Massachusetts, USA, 1997. http://www.meitca.com//HSL/Projects/Concordia/ /MobileAgentConf_for_web.htm

[22] Harrison, C. G., David M. Chess, D. M., Kershenbaum, A., "Mobile agents: Are they a good idea?, " Technical report, IBM T.J. Watson Research Center, Yorktown Heights, New York, USA, 1995. *http://www.research.ibm.com/massive/mobag.ps.*

[23] Green, S., Hurst, L., Nangle, B., Cunningham, P., "Software Agents: A review, " Trinity College, Dublin, Ireland, 1997. http://www.cs.tcd.ie/research_groups/aig/ /iag/*/pubreview.ps.gz*

Appendix A: Comparison

Agent System	Aglets	Concordia	Odyssey	Voyager
GUI	Yes	Yes	No	Yes
Modular design	No	Yes	No	No
Mobility mechanism	Sockets	RMI	RMI	RMI
Persistence	None	Implicit	None	Explicit
Security	Security Manager	Security Manager	Java based	Restricted operations
Direct agent-agent comm.	Yes	No	No	Yes
Comm. types provided	Sync Async Broadcast	Group events Filtered events Collaboration	None	Sync Async Mailbox Broadcast