Technology Review of Java-based Mobile Agent Platforms

Jens Krause[1,2]

Abstract

The concept of mobile agents promises new ways of designing applications that better use the resources and services of computer systems and networks. For example, moving a program (e.g. search engine) to a resource (e.g. database) can save a lot of bandwidth and can be an enabling factor for applications which otherwise would not be practical due to network latency. Leveraging on the strengths of Java, several Java-based mobile agent platforms became available recently. This report introduces features Java-based mobile agent platforms should provide in our view and investigates in detail the current versions of three existing platforms. The aim is to provide developers with a detailed comparison and to help in selecting an appropriate platform.

Keywords: Mobile Agents, Mobile Code, Platforms, Java

[1] IBM Zurich Research Laboratory; mailto: jkr@zurich.ibm.com
[2] Swiss Federal Institute of Technology Lausanne (EPFL), DI-ICA; mailto: Jens.Krause@ica.epfl.ch

1 Introduction

A mobile agent needs an infrastructure in order to be transported over the network, to contact and communicate with other mobile agents, to exploit resources (CPU cycles, memory, persistent storage, databases, etc.), in order to accomplish its assigned task. This infrastructure is provided by a *mobile agent platform*. The mobile agent platform further provides support for agent development, agent testing and the management of agents.

Mobile agent platforms have been implemented in existing programming languages (C, C++), as extensions to existing programming languages (Agent-Tcl [Gray95]) and completely new programming languages (Obliq [Card95], Telescript [Gene95]). Most of these platforms did not evolve beyond an experimental, academic stage. General Magic was the first company to offer a mobile agent platform called Telescript for industrial use. One reason for its failure was the proprietary nature of the Telescript language.

In the past 2 years, early versions of Java-based mobile agent platforms have been made available on the Internet. They take advantage of the strengths of Java [Java97], such as the availability of the Java Virtual Machine on almost every operating system and computing platform, integrated networking support, the serialization API, security features and interfaces to existing technologies like open database connectivity (ODBC, [Micr97]), common object request broker architecture (CORBA, [YaDu96]) and simple network management protocol (SNMP [SFDC90]).

The aim of this review is to investigate Java-based mobile agent platforms available today for their suitability, and to help in selecting the right platform. In Section 2, we define components common to most mobile agent platforms, while Section 3 introduces features which we believe a mobile agent platform should support. Section 4 performs a pre-selection in order to limit the number of platforms to investigate in detail throughout Section 5. The last section provides a summary.

2 Common components

Java-based mobile agent platforms are a recent and fast evolving technology. Therefore, it is not surprising that different terms are used for the same functional components of different platforms. This section defines the terms used throughout this review for components common to all platforms (see Figure 2.1). We stay here closely to the definitions of the Object Management Group's (OMG) Mobile Agent Facility (MAF, [OMG97]) since this is a first attempt to standardize certain parts of mobile agent platforms in order to achieve interoperability.

The mapping to the platform-dependent terms are given during the investigation of the actual platforms in Section 5. Implementation aspects are mentioned if there is a standard way in Java to implement a certain functionality.

A (mobile) *agent-enabled network* comprises of *computational devices*, usually computers but also personal digital assistants (PDAs) or other devices that provide a Java Virtual Machine (JVM). The computational devices are interconnected through a networking infrastructure, usually the Internet or an intranet. Interconnected does not imply continuous connectivity as devices such as PDAs might be temporarily disconnected.

A computational device may host one or more *agent systems*. An agent system provides a computational infrastructure, offers access to local resources, and takes care of agent transport and communication across the network. It can be seen as a layer between the JVM and the places (see

below). An agent system can be identified through its network address, e.g. an IP address and a port number of a transport protocol (usually TCP). An agent system might be realized as a standalone application, or embedded in an already existing application or an Applet.

A *place* is a stationary object that provides a uniform execution environment for agents. A place offers a means for maintaining and managing running agents and applies security policies to protect the agent system and the computational device against malicious agents. One agent system may serve multiple places.

A *mobile agent* is a Java program that acts autonomously on behalf of a person or organization. A dedicated execution thread enables it to perform tasks on its own initiative. A mobile agent has the unique ability to transport itself from one place to another even if the places are located at different agent systems and different computational devices. In order to move, a mobile agent is marshaled into a byte stream using Java object serialization, than transported (over the network) to another place, and finally unmarshaled. It then resumes execution on the remote place.

Every agent needs to have an *identifier* (ID) through which it can be identified for management operations, and can be located via a directory service. The ID has to be unique between different places, agent systems and computational devices to allow an unambiguous addressing even after the agent has moved.

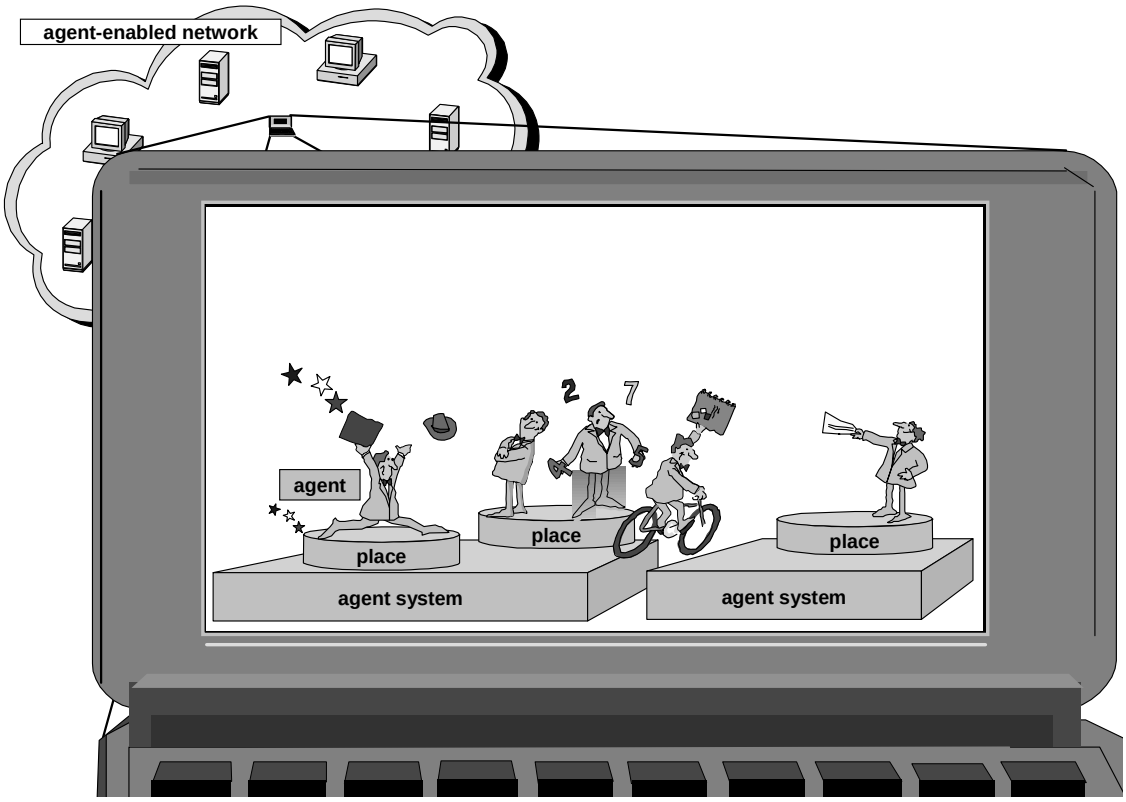

Figure 2.1: Common components

3 Features

After presenting the common components and concepts in the previous section, we now move on to introduce individual features that are used later to distinguish the investigated mobile agent platforms.

3.1 Transport

The transportation facility of a mobile agent platform provides the means for an agent to move from one place to another across the network. It also takes care of delivering remote messages and events and supports the remote invocation of methods. Its underlying network infrastructure might be composed of unreliable transmission media and temporarily disconnected network nodes or subnetworks. The transport mechanism of a mobile agent platform has to cope with these aspects.

Java-based mobile agent platforms usually exploit the Java object serialization mechanism. However, this mechanism allows only the serialization of the agent attribute values and not the execution stack. Therefore, data from the execution stack has to be transferred to agent attributes before an agent gets transferred to a different place. A mobile agent developer has to keep this in mind because it's his responsibility to determine and save the required data from the execution stack that is needed to be transferred with the agent. A Java-based mobile agent platform that provides automatic extraction of the execution stack needs to modify the JVM and therefore looses Java's run-everywhere property.

3.2 Communication

Communication is a fundamental property of an agent. Without communication no interaction with other agents or with the agent environment is possible. Therefore, a mobile agent platform is required to provide a distributed communication infrastructure.

The goal of communication is the exchange of information. Among the agent systems available today we noticed basically three kinds of operations during which information is exchanged: message passing, event distribution and method invocation.

Messages can be divided into *synchronous messages* which block, and *asynchronous messages* which do not block the sender during message transmission. If a synchronous message is sent, the execution thread of the sender is blocked until a reply is received. Synchronous messages are usually employed in situations where the sender cannot continue execution until the message is delivered and a response received. They can also be used to synchronize a set of agents. An agent will use an asynchronous message if it does not want to delay other tasks until a reply for the message has been received. In this case, the sender usually keeps a reference to the message, and checks at a later point in time whether a reply has been received or not. Asynchronous messages are therefore sometimes also called *future messages*.

Orthogonal to the above classification is the differentiation of messages according to the number of recipients of a message. If the message has only one recipient, we speak of a unicast message. Unicast messages are often the default case, so the unicast nature of the message is not explicitly mentioned. Messages that are sent to a group of selected recipients are called multicast messages. Broadcast messages are delivered to all possible recipients. A specialization of the multicast message scheme is the so-called publisher/subscriber scheme. Here, agents can subscribe to an information source (the publisher) to a subject or, more generally, to a message type they are interested in. Messages are than delivered to all corresponding subscribers.

Another distinguishing criterion for messages is whether a message is delivered to an agent at the same place or not. Messages that do not leave the place of the sender are called local messages. If the receiver is located at a different place, a remote message has to be used. Local messages are usually faster than remote messages.

The ability to assign priorities to messages is another useful feature. Using priorities, one can give preference to important messages and have them delivered faster than less important messages.

Events provide information about occurrences that happen in some component of the system. Examples are a *place event* like the arrival of a new agent at a place, or a *user interface event* like a mouse click. The state of the art in Java programming is the delegation event model. It was first introduced by Java Beans and has now also been adopted by the Abstract Windowing Toolkit (AWT). With this model, an agent interested in some event of a component registers itself with the component. In order to receive events, it has to implement an event handler (called “adapter”) with a corresponding listener interface for the event. If an event occurs it will be delivered (delegated) to all registered adapters. As with messages we can again distinguish between *local events* that stay at the place where they occurred, and *remote events* that are delivered to a different place.

The third possibility for agents to exchange information is *method invocation*. Here the information is passed as an argument to a method of an agent or as the return value of a method. Again we can distinguish between a method invocation that is carried out at the same place (*local*) or between different places (*remote*).

3.3 Persistence

Agent persistence or, more generally, object persistence is required to ensure that agents and other vital components of the agent-enabled network such as the directory service, survive system crashes. The usual way for Java-based mobile agent platforms to implement persistence is to take advantage of the Java object serialization mechanism. The obtained byte stream image can be easily written to nonvolatile storage like a hard disk. Persistence improves the robustness and fault tolerance of a mobile agent platform.

There are different levels of persistence. The simplest level is agent-invoked checkpointing. This means that the agent decides by itself to produce a persistent image after it has finished some task and has reached a consistent state. A higher level of robustness can be obtained if the agent system automatically produces images before and/or after major events during the life cycle of an agent. Such events might be the creation or cloning of an agent or transportation to another place.

3.4 Directory service

One of the benefits expected from a mobile agent platform is the efficient usage of distributed resources. Resources might be CPU cycles, memory, secondary storage or other input/output facilities. A place usually provides access to local resources, or hosts other objects that provide access to these resources. While these are usually static resources, a mobile agent providing some service can be seen as a dynamic resource. The *directory service* gives support for retrieving these resources in the agent-enabled network.

The directory service allows registration of a symbolic name together with some reference to the object that manages the resource. The symbolic names are usually arranged in a hierarchical tree. Better scalability can be achieved if the subtrees of the directory are distributed across the network. This might also increase performance if the subtree containing information about resources is kept at the same location as, or close to, the resources itself.

A special challenge to the directory service is the handling of mobile objects or agents. The agent programmer’s task is simplified if the directory service traces the moves of mobile objects automatically and keeps references to them up to date.

3.5 Security

"A piece of software that executes on a computer where it manipulates its execution environment as well as other active components. It decides by itself about when to multiply itself, where to go

and when to die." - What at a first inspection might appear to describe a virus is actually an informal description of a mobile agent!

An agent-enabled network is exposed to the same threats that any open and distributed system has to struggle with: disclosure, modification, denial of use, misuse, abuse, and repudiation. Agents themselves are also exposed to these threats, not only when they travel, but also when they are at a malicious agent system or place (see [KLO97]).

A mobile agent platform has to provide security mechanisms that detect and prevent malicious actions. Without strong security mechanisms, a mobile-agent system will justifiably never be accepted and used. [Gray96] identified four interrelated problems for a mobile agent-enabled network:

- protection of an agent system against malicious agents,
- protection of an agent against malicious agents,
- protection of an agent against malicious agent systems,
- protection of the agent-enabled network as a whole.

All of these considerations have been discussed in the mobile agent literature [LeOu95, CGH+95, TaVa96] but only for the first two serious solutions have been found. These are introduced below.

1. *Protection of an agent system against malicious agents:* Java introduces the “sandbox” concept which uses a security manager to protect the system against untrusted code (e.g., Applets downloaded from the Internet). A security manager uses hooks in the Java library methods to intercept a method call that might expose the system to a security threat. The security manager can then decide to allow or reject the method call, depending on the security policy that has been implemented. This is typically done for methods providing access to the file system or the network. But also other APIs that provide access to resources like the Java Database Connectivity (JDBC), Remote Method Invocation (RMI), Java Native Method Interface (JNI) or ActiveX have to be monitored.
2. *Protection of an agent against malicious agents:* this problem might be addressed in a similar way as the previous point. Here, the security manager, with help from the agent system and/or the place, might be able to intercept calls to methods of an agent by other agents.

Only these two points are considered throughout the detailed examination in Section 5.

3.6 Integrating agent systems

The goal of a mobile agent platform is to provide the programmer with convenient means to use code mobility for application development. However, there are many applications that do not need code mobility to solve most of their problems because they can be implemented efficiently using conventional programming techniques. In these cases one might not want to implement the whole application as a set of agents. Instead, one may want to *integrate* a small agent system within a bigger application that provides an interface or entry point where services are offered to agents. In this case, we speak of an *embedded agent system* as opposed to the *standalone agent system* which is usually provided by the manufacturer of the mobile agent platform.

If the agent system itself is subject to frequent changes or has to be distributed to a large number of machines, a solution could be to *embed the agent system in a Java Applet* that easily can be downloaded from an HTTP server using a Web browser.

However, this case is a tricky one. Usually, if the Applet security manager [Java97a] is not turned off, Applets are not allowed to access the local file system or establish network connections

to hosts other than the HTTP server from which they have been downloaded. The solution is to install a kind of software router or an agent system acting as a software router on the HTTP server that provides the connectivity the Applet cannot get on its local host. The software router forwards agents and messages from and to the embedded agent system in the Applet. However, access to local files is still a problem.[3]

3.7 Administration

The administration of an agent-enabled network includes a variety of tasks. Agent systems have to be started, stopped, restarted and upgraded. An administrator might also want to monitor the activities of certain agents and needs the means to intervene (create, delete, suspend, resume, or move agents). Monitoring the performance of the agent system allows the administrator to better distribute the load across the network.

Furthermore the persistent storage might be subject to management activities (increase/reduce allocated disk space, free resources). Another management aspect is security. Here keys of trusted parties have to be distributed or retracted, and security policies need be updated.

A graphical user interface (GUI) allows user interaction with the agent system and the agents on it. However, as the number of agent systems increases, the need for remote administration emerges. We can also think of computational devices that host agent systems but do not have graphical displays or attached terminals. Here remote administration is of particular importance.

Another task for the agent system is to provide support for debugging. Even parallel processes on the same machine are difficult to debug. Things become worse when these processes start to roam the network. A logging facility can help to trace the moves and activities of agents. However, such a logging facility is of little help if logs are spread over a dozen or more agent systems and have to be investigated manually in order to reconstruct what happened with a certain agent. The agent system should provide a means for the agent to always direct its traces always to the same destination.

3.8 Usability

This feature brings together system requirements, installation support and documentation. Low system requirements are important for the success of a middleware like a mobile agent platform. It should be easy to fulfill the system requirements in order to operate agent systems on a large number of different computational devices. Typical requirements for a Java-based mobile agent platform are the availability of the latest version of the Java Development Kit (to date JDK 1.1) for the computational device, and network connectivity (e.g. a TCP/IP stack) at the computational device.

A simple and clear installation procedure together with sample agents that demonstrate how to exploit the features of the mobile agent platform ease the start for the beginner. Last but not least, a good documentation is important for a mobile agent platform as it is important for any software development environment.

4 Pre-selection

The aim of this review is to provide a decision base for the selection of the appropriate platform to software developers intending to start with mobile agent programming. To date, a large number of platforms that allow mobile agent development can already be found on the Internet. Therefore,

[3] These restriction are relaxed with the JDK 1.1 code-signing facility.

we have initially chosen 6 platforms that looked to us as the most promising to limit the size of this report. More Java-based mobile agent platforms can be found in Table 4.2.

The 6 platforms that have been chosen are presented briefly (in alphabetical order) below. A pre-selection is carried out before the detailed discussion of the selected platforms follows. Criteria for the pre-selection are the compliance with the latest JDK version and the free availability of the platform, at least for noncommercial use.

4.1 Aglets Workbench by IBM

The IBM Aglets Workbench [LOKK97] was one of the first mobile agent platforms fully implemented in Java. The Aglets Workbench is intended to provide another Java component model like Java Applets or Java Beans. This also explains the name Aglet which is a new word derived from agent and Applet.

The first alpha version of the Aglets Workbench was released in July 1996. The version investigated in this report is Alpha 5c that is also supposed to be the last alpha version. The Aglets Workbench runs with JDK 1.1 and is free available for noncommercial use. Therefore it fulfills the criteria for the pre-selection and is examined in detail in Section 5.

4.2 Ara from the University of Kaiserslautern

Ara is a multi-language mobile agent platform [PeSt97]. Its architecture consists of a core (the agent system) in which modified interpreters for different programming languages can reside. The modified interpreters provide the contexts (places) for agents written in the corresponding programming language. Agent interaction is possible between agents implemented in different programming languages. The core provides inter-agent communication, a directory service, persistence, agent transport (with execution state) and basic security features.

The retrieval of the execution state is a unique feature of Ara. However, this requires modifications in the language interpreters. The interpreter for a given language has to be extended with functions for program control and execution state saving. Modified interpreters for Tcl (version 7.4) and C/C++ by means of pre-compilation to an interpretable byte code are available. The announced Java interpreter was not available at the time this report was written. This excluded Ara from a detailed examination.

4.3 Concordia by Mitsubishi Electric ITA

Concordia was developed by the Mitsubishi Electric Information Technology Center America (MEITA). Mitsubishi emphasizes the clear modular structure of the Concordia platform [WPW+]. Concordia uses a modular architecture that assigns different functionality to 5 managers within the agent system, called the Concordia server [MEIT97]:

- The event manager handles registration, posting and notification to and from agents.
- The persistence manager maintains the state of agents in transit around the network and allows to checkpoint and restart agents.
- The queue manager is responsible for the scheduling and possibly for retrying the migration of agents between agent systems.
- The administration manager allows to manage all of the services provided by Concordia, including the different managers. It supports remote administration from a central location, and provides a user interface.

- The security manager is responsible for identifying users, authenticating their agents, protecting server resources and ensuring the security and integrity of agents.

At this point in time Concordia is not freely available. Mitsubishi Electric ITA requires users to sign a nondisclosure agreement to obtain the current version Beta 2.0. Therefore it was not possible to take a closer look at this platform, despite the interesting features it offers.

4.4 Mole by the University of Stuttgart

The current version of Mole [SBH96], [BHSR97] is Alpha 2.0. The agent system in Mole, called *engine*, can provide multiple places called locations. Mole uses a proprietary implementation of Sun RPC in Java for the transport of agents between different locations. Mole relies on JDK 1.0.2 together with the Sun objio-package v0.8 (see [Mole97]). Documentation is available only for the preceding version Alpha 1.0. The javadoc description of the different classes and error messages of the Mole engine (agent system) are also very limited which makes mobile agent development difficult. Therefore we delayed the examination of the Mole Platform until the version Alpha 2.1 is available which also will run with JDK 1.1.

4.5 Odyssey by General Magic

General Magic was the first company to attempt a commercialization of a mobile agent platform with its Telescript language. At that time General Magic was granted a patent for mobile agent technology. However, this project failed because of the propriety nature of the programming language and their marketing policy. Early 1997, General Magic started to re-implement the Telescript architecture in Java. The new platform is now called Odyssey [Odyssey].

A unique feature of Odyssey is that it allows deployment of different agent transport mechanisms. The technologies that are supported are Java Remote Method Invocation (RMI), Microsoft's Distributed Component Object Model (DCOM), and the Internet Inter ORB Protocol (IIOP). The current version is 1.0 Beta 2 and relies on JDK 1.1. A detailed examination is carried out in the next section.

4.6 Voyager by ObjectSpace

ObjectSpace defines Voyager as an "Agent ORB" (Object Request Broker) [Voyager]. The idea here is to add the functionality of object mobility to an ORB middleware. Within this architecture, a mobile agent is seen as an autonomous mobile object. Unique to Voyager is its architecture for message and event multicasting, called "Space". It has been constructed to provide efficient and fault tolerant multicast message forwarding for large numbers of recipients.

The current production release of Voyager (version 1.0.1) relies on JDK 1.1.x and is free for noncommercial and commercial use with only few exceptions (e.g. implementation of a middleware on top of Voyager). We will discuss Voyager in the next section.

4.7 Summary

Table 4.1 summarizes the Java-based mobile agent platforms of the pre-selection. The selected platforms are discussed in the following section. Pointers to more Java-based mobile agent platforms are given in Table 4.2.

Agent platform	Aglets	Ara	Concordia	Mole	Odyssey	Voyager
Manufacturer	IBM	Univ. of Kaisers-lautern	Mitsubishi Electric ITA	Univ. of Stuttgart	General Magic	ObjectSpace
JDK version	1.1	N/A	1.1	1.0.2	1.1	1.1
Availability	free for non-commercial use	free for non-commercial use	non-disclosure agreement	free for non-commercial use	free for non-commercial use	free for commercial use
Selected	Yes	No	No	No	Yes	Yes

Table 4.1: Java-based mobile agent platforms in pre-selection

Agent platform	Manufacturer and URL
Astrolog	IRISA/SOLIDOR: http://www.irisa.fr/solidor/work/astrolog.html
JavaNetAgents	ONERA (Office National d'Études et de Recherches Aérospatiales): http://www-laforia.ibp.fr/~merlat/jna-web/jna.htm
JavaToGo	University of California at Berkeley: http://ptolemy.eecs.berkeley.edu/dgm/javatools/java-to-go/
Mobile Unstructured Business ObjecT (MuBot)	Crystaliz: http://www.crystaliz.com/logicware/mubot.html

Table 4.2: More Java-based mobile agent platforms

5 Detailed Examination

In the previous section, only 3 of the initially 6 mobile agent platforms could pass the pre-selection. Now, the selected platforms are discussed in detail in the following sections. For the discussion, we apply the same analysis scheme to each platform. After a short introduction, we map the terms for the common components defined in section 2 to the terms used in the actual platform. Platform specific differences are pointed out as well. Then, we work through the service features introduced in Section 3. An overview of the discussed platforms is presented at the end of this section.

5.1 IBM Aglets Workbench

IBM defines the Aglets Workbench as an environment for building network-based applications that use mobile agents to search for, access, and manage corporate data and other information [Aglets]. The Aglets Workbench is intended as an addition to Java component models for executable content, in particular the Java Applet and Java Servlet models. The version examined in this report is Alpha 5c.

Mapping of terms for common components

In the Aglets Workbench, the agent system is referred to as the *Aglet server*. The implementation of an Aglet server is called *Tahiti*; it provides a graphical user interface. An Aglet server provides only one place called *Aglet context*. Nevertheless, a computational device can host multiple Aglet servers running in different Java Virtual Machines. An Aglet server / Aglet context can be uniquely addressed by a host name and a TCP port number.

In the Aglets Workbench, agents are called Aglets like the workbench itself. An Aglet is network-wide uniquely identified by its `AgletID` that is generated by the Aglet context.

Transport

The Agent Transport Protocol (ATP) is the primary transport mechanism used in the Aglets Workbench to transfer Aglets and messages between Aglet servers/contexts. The ATP specification is modeled on the HTTP/1.0 [BFN96] specification. ATP offers the opportunity to handle mobility of agents in a general and uniform way [LaAr97]. The protocol is proposed as an open standard and might also be adopted by other mobile agent platforms.

ATP requests can also be embedded in HTTP requests. This is called HTTP tunneling and, in combination with a HTTP proxy server, enables Aglets to cross firewalls.

Communication

For location transparency in communication the concept of an *Aglet proxy* has been introduced. An Aglet proxy forwards messages to or from the actual Aglet context of its agent. To communicate with an other Aglet an Aglet's proxy is required regardless whether the Aglet is at the same Aglet context or not.

The Aglets Workbench only supports messages with reply. Unicast messages can be either synchronous or asynchronous. The addressee of a message can send any kind of object as a reply. If an Aglet is not able or, for some reason, not willing to handle a message, it can delegate the message to another Aglet. This happens transparently to the sender, which will not recognize that the reply actually came from an Aglet other than the intended recipient.

In case of an asynchronous message, the sender receives a `FutureReply` object. This object can be checked later to verify whether a reply has been received or not. The agent can also decide to block itself until a reply has been received. Unicast messages can either be local or remote.

In the Aglets Workbench, multicast messages are sent to the Aglet context. However, the Aglet context distributes messages only locally. Multicast messages are implemented as asynchronous messages. After a multicast message has been handed over to the Aglet context, the Aglet context returns a `ReplySet` back to the sender. This allows successive handling of the incoming replies.

A special feature is the MessageManager which controls incoming messages. Each message type can have a priority and will be queued in the message queue in accordance with its priority.

The Aglets Workbench supports a delegation event model similar to that used in Abstract Windowing Toolkit (AWT) of JDK 1.1 and Java Beans [Sun97]. In order to be able to receive a certain event type, an Aglet has to implement the corresponding event interface and event listener (adopter) class. To actually receive events, the adopter has to be registered with the event source. For example, an event source might be the Aglet context for context events, a component of the GUI or another Aglet. Like multicast messages, events are distributed only within the same context.

The Aglets proxy prevents other Aglets from directly accessing an Aglet's public methods. Thus, this communication channel cannot be used by Aglets. However, Aglets justify this with the obtained security level (see below).

Persistence

The Aglets Workbench provides agent-invoked checkpointing. A so-called snapshot of the current state of an Aglet can be saved in the "spool" directory. The snapshot will be activated only if the Aglet is accidentally killed, e.g. because of a system crash. The snapshot will be removed if the Aglet is deactivated, moves to a different context, or finishes execution.

Directory service

To retrieve other Aglets, an Aglet can obtain a list of Aglet proxies for Aglets hosted by the context. This list can be obtained only from the current, local context. To get information about which Aglets are currently in a remote Aglet context, an Aglet is required to move and query the context locally.

A distributed directory service can be realized using the Registry/Registrant pattern. A Registry maps symbolic names to URLs. This allows to locate an Aglet in the Aglet-enabled network if it has been registered at the Registry beforehand. In order to keep the Registry up to date, an Aglet has to either inherit form the abstract Registrant class, or manually update the URL at the Registry after it has moved.

Aglet proxies are not updated automatically. If an Aglet moves to a different context all proxies for that Aglet become invalid. In order to update the proxy, the Aglet context has to be queried, specifying the `AgletID` and the URL of the new location.

Security

In Section 3, two security aspects have been introduced that can be treated by Java-based agent platforms.

The first problem was to protect the agent system against malicious agents. For this, the Aglets Workbench uses the Java concept of a security manager. The current version distinguishes only between trusted Aglets that have been loaded from the `CLASSPATH`, and untrusted Aglets that have been loaded via the network, e.g. from another Aglet context or from a remote code store (FTP or HTTP server). Different security policies can be configured for trusted and untrusted Aglets. Restrictions that can be defined in a security policy cover the following points:

- Access to the file system for reading and/or writing to subtrees of the directory tree.
- Network access for listening on ports and/or connecting to hosts (IP address or DNS name).
- RMI usage as client and/or server.
- Access to JDBC.
- Creation of windows on the screen.

Native method calls from Aglets are prevented by the `AgletsSecurityManager`.

In [KLO97], a fine-grained security model has been introduced for the Aglets Workbench. The model proposes secure agent transport over the network using state-of-the-art protocols and authentication mechanisms. This allows granting of more access permissions to Aglets received from trusted parties than one would otherwise give to untrusted Aglets. This security model is expected to be integrated in the next releases.

The second security problem, the protection of an agent from malicious agents is addressed with the Aglet proxy concept. An Aglet proxy forms a shield around the Aglet and protects so the Aglet from direct access to its public methods [Lang97]. This excludes a lot of possible attacks. An attacker who wants to interfere with an Aglet can only use the communication channels for messages or events. The Aglets Workbench does not provide special security measures to secure these channels. Hence, it is the responsibility of the Aglet programmer to determine the identity of the originator of messages and events, and to protect the message content if this is needed for a specific application.

Integrating agent systems

The default agent system for Aglets is called Tahiti. It provides a graphical user interface that allows the user to control and interact with the Aglets on that Aglet server (for a more detailed description see the point “Administration” below.) A command line version of this Aglet server is also available.

The Aglet Server Development API allows embedding of an Aglet server in a Java application. If the full Aglet server functionality is not needed, and the application instead only wishes to control and communicate with Aglets on an Aglet server, the lightweight Aglet Client API can be used.

The workbench also supports HTTP message handling. This adds a Servlet-like functionality to receive an HTTP/CGI request from a Web browser and send HTML pages back to the browser.

Applet-to-application connectivity is provided with the Fiji add-on package. Fiji allows integrating of an Aglet server within an Applet. On the HTTP sever which stores the Applet, a router daemon forwards incoming and outgoing messages and agents. Two versions of this router are provided.

The Servlet version of the router can be automatically started if an Servlet-enabled HTTP server is used. Should the HTTP server not support Servlets, the standalone version of the router has to be started by the administrator of the HTTP server.

While messages can be sent to standalone Aglet servers as well as to Applet-contained Aglet servers, Aglets can be sent only to standalone Aglet servers. However, the `retract()` method allows pulling back of an Aglet that has left an Applet-contained Aglet server.

Administration

The main purpose of the graphical user interface of the Aglet server (Tahiti) is to control and interact with Aglets in that server’s Aglet context. Aglets can be created, disposed of, cloned, dispatched to and retracted from another Aglet server, as well as deactivated (saved to second storage) and activated. The dialog function is available to start a user interaction with the Aglet if the Aglet has implemented this function (usually a window is opened).

Also are provided are a view functions for server management and debugging. Memory usage and the log file can be inspected. It is also possible to get a list of active threads and switch the debugging mode on or off for the server.

The Aglets Workbench does not provide a tool to remotely manage Aglet servers. However, the Aglet Client API or the HTTP message handling might be exploited to implement such a functionality.

Usability

The investigated version (Alpha 5c) requires JDK 1.0.2 with the RMI prerelease or JDK 1.1. A set-up utility is provided for installation. For documentation an installation guide and a “getting

started" guide are provided. A set of example Aglets eases the learning phase for the beginner. Besides the usual javadoc API documentation the unfinished Aglets Cookbook explains basic concepts of the Aglets Workbench.

5.2 General Magic: Odyssey

General Magic is a pioneer in the development of mobile agent platforms. With its Telescript language, General Magic implemented the first commercial mobile agent platform in 1994. However, Telescript missed commercial success because of the propriety of the language and General Magic's licensing policy. In 1996, a first Java project called Tabriz was launched; but it was withdrawn from the market only a few months later. Odyssey is the name of a new project that tries to reuse the concepts of the Telescript platform in Java. It was announced early 1997 [Odyssey].

Mapping of terms for common components

Odyssey uses the term *agent system* consistent with our definition in Section 2 (the class that implements an agent system is called `Engine`). An agent system can provide multiple *places*. This saves resources in case a larger number of places are running on the same machine since it requires only one JVM and therefore only one system process.

The `BootPlace`, the initial place created when the agent system is started is distinct from all other places. When the BootPlace terminates, the agent system shuts down, terminating all agents and places currently existing within that agent system [Gene97].

Specific to Odyssey is that both agents and places inherit from the same abstract class called `OdysseyProcess`. The behavior of a process is defined in its `life()` method. Processes can be uniquely identified by a `ProcessName`.

The Odyssey Worker is a special implementation of an agent. A worker is structured as a list of tasks with associated destinations. This itinerary will be carried out sequentially.

Transport

A unique feature of Odyssey is that it allows the deployment of different agent transport mechanisms. General Magic states that the Odyssey agent system can use any reliable transport service to move agents from one system to another. By default, Odyssey uses RMI. The `Transport` interface gives programmers the possibility to customize the agent transport mechanism of Odyssey. Odyssey provides interfaces to DCOM (Microsoft's Distributed Component Object Model) and IIOP (Internet Inter-ORB Protocol).

Odyssey does not support the class loading from one agent system to another. For example, if RMI is used as the transport mechanism, one can specify a `CODEBASE` (see below section: Agent system) from where to load class files that are not in the local `CLASSPATH`. However, an agent whose class files are not available from one of these two sources can not enter this agent system. As a result, the administrator of the agent system has to know which agents might want to visit an agent system before the agent system is started.

Communication

General Magic has implemented a simple mobile agent platform. Odyssey does not have messages and events. In Odyssey, an agent travels to some place and holds a meeting with another agent. During a meeting, the initiator of the meeting communicates with its partner through direct method calls on the partner.

Persistence

Odyssey does not support agent persistence.

Directory service

Odyssey's directory mechanism allows retrieval of places. The `Finder` interface has to be implemented by application designers that wish to interface the directory mechanism used by the Odyssey agent system with an existing directory mechanism. By default, the `RMIFinder` is installed that provides an interface to the `rmiregistry`.

The static class `Published` offers a mechanism for exchanging references to public objects. This class performs a mapping from a name in string form to an object. The names do not form a directory tree. The class Published can be accessed only from places within the same engine (agent system). Registered objects which might also be agents are not supposed to move to a different agent system.

Security

The set of precompiled class files delivered with Odyssey also contains a class called `OdysseySecurityException`. This leads us to believe that some Applet-like trusted/untrusted security scheme has been implemented, but the lack of documentation has not allowed to test this feature.

Integrating agent systems

The standalone agent system of Odyssey is controlled through a configuration file. In the configuration file, the `BootPlace` and some options, e.g., transport mechanism and finder to be used, can be specified. After the engine has been started, the front panel basically presents a list of active threads, and allows termination of the engine.

The class `OdysseyApplet` claims to provide means to bootstrap a browser-based Odyssey agent system. Again, due to the lack of documentation or examples it could not be tested and verified. Also no mechanism could be identified to embed an Odyssey agent system in a Java application.

Administration

Odyssey provides only very limited tools for administration. All that can be done interactively on the local engine is to view active threads, the Odyssey version number and to quit the engine.

The `AuditTrail` class provides a uniform means to report debugging information. Three different types of logging messages are available: normal messages, important messages and exceptions. In addition the amount of logging information can be defined in the boot configuration file. Each logging message in the Odyssey system has been assigned to one of the following categories: process, travel, meeting, transport, finder, or worker. These logging messages can be directed to the standard output stream, a separate window or a file. If logging information is sent to a file, the agent can no longer move since a file descriptor is not serializable.

Usability

Odyssey requires JDK 1.1. To support the Finder, which locates destination places, an RMIRegistry has to run is required. The installation is simple, but the documentation is currently limited to some example agents, a short FAQ list and the usual javadoc-generated API

documentation. As an indication of the lack of documentation, we note that none of the examples cover the agent meeting, an essential mechanism of Odyssey.

5.3 ObjectSpace: Voyager

ObjectSpace has chosen a different approach than the other two mobile agent platforms investigated before. Voyager aims at providing an alternative middleware to CORBA and RMI. The advantage over these technologies is object mobility. Voyager allows, in contrast to previously discussed platforms, to move also objects not only agents. Thus, a mobile agent is seen as an autonomous mobile object.

Mapping of terms for common components

In the Voyager architecture, an agent system is called a *Voyager server*. There is no notion of places in Voyager. The functionality of places is integrated in the Voyager server.

Transport

Voyager relies on the Java object serialization mechanism to transform objects into byte streams. A proprietary protocol (layered on top of TCP) is used to transport the serialized object across the network. For persistent objects, a reliable transport can be guaranteed against server crashes.

Communication

The key vehicle for communication in Voyager are *virtual references*. A virtual reference is defined as a representative for an object not only for agents. Virtual references are distinct for objects from different classes. For every class whose objects are intended to be remote referenced (called a *remote-enabled* class), a so-called *virtual class* (similar to a RMI stub class) has to be created. A virtual class is derived from the class in question, using the virtual class creator (`vcc`, pendant to `rmic` of RMI). `Vcc` performs a kind of post-compilation which remote-enables classes for which only Java class files but no source code is available. Virtual references stay valid even when the object moves. This is achieved through the use of *forwarders*. Forwarders remain behind when an object that has virtual references moves. The virtual reference gets automatically updated with the reply to the next message it passes on.

Voyager uses `Messenger` agents for message delivery. A Voyager message consists of a method name and a list of parameters. The method name, that is given in string form, is mapped to the receiver's corresponding method. Hence, a Voyager message always results in a method call on the receiver object.

Three different types of unicast messages are available. By default, a synchronous `Messenger` (called `Sync`) is used but asynchronous (called `Future`) and `OneWay` messengers are also provided.

If messages are sent with the `Future` messenger, the sender receives a `Result` object which can later be checked for a reply. If the sender decides that it is no longer interested in the reply to a Future message, a call to the `kill()` method of the `Result` object prevents the messenger from delivering back the reply if it has not already been done.

Unique to Voyager is its architecture for message and event multicasting called Space. It is designed to provide efficient and fault-tolerant multicast message forwarding for a large number of recipients. Within the Space architecture, groups of possible recipients are united in so-called subspaces. The subspaces are linked together to form a larger logical group, or Space. Each message sent into one of the subspaces is cloned to each of the neighboring subspaces, before being delivered to every object in the local subspace. This results in a rapid, parallel fanout of the message to every

object in the Space. A mechanism in each subspace ensures that no message or event is processed more than once, regardless of how the subspaces are linked together [Obje97].

Multicast messages are delivered within Voyager by means of the Space architecture. OneWay messengers are used to deliver messages to the objects in a Space. The sender can add a so-called Selector to define the criteria that the receiving object has to fulfill in order to receive the message.

An implementation of the Selector is also used in the publisher/subscriber message scheme. Agents/objects can register their interest in certain topics or a whole subtree of topics at the information source (the publisher). Updates for a topic are delivered to all subscribers using oneway multicast messages and in the Voyager Space architecture.

Voyager follows the standard Java Beans event model for event and listener syntax and semantics. There are three different event types corresponding to the event source: `ObjectEvent`, `SystemEvent` and `SpaceEvent`. Events are distributed using the Voyager Space architecture. This also allows to distribute Java Beans events to remote agent systems.

As described above, Voyager messages result in method invocations on the receiver object. To speed up communication, an agent/object can move itself to the agent/object it wants to communicate with, and then locally invoke a method directly on the agent/object.

Persistence

Voyager supports database-independent distributed object/agent persistence. By default the class `VoyagerDb` is used to stores objects. But any database may implement the `Db` interface and can then also be employed for this purpose.

An object can be made persistent through a call to the `setPersistant()` method. This also causes the object to be saved immediately. Subsequent checkpoints of an object can be stored by invoking the `saveNow()` method. Objects can also be temporarily flushed to the database to save resources.

If a persistent object/agent moves to another Voyager program, it gets automatically saved at its new location and its persistent image is removed from its former location.

Directory service

Voyager allows the registration of an object in a distributed hierarchical directory structure provided by the Federated Directory Service. Subtrees of this directory structure can be spread across the Voyager-enabled network. This minimizes the single-server bottleneck and point-of-failure associated with monolithic directory services. The Voyager directory service allows registering of remote and mobile objects. The virtual references stored in the directory tree are updated automatically when an object moves using the concept of forwarders (see section communication above).

Security

Voyager addresses only the first of the security aspects introduced in Section 3: the protection of the agent system against malicious agents. This is done by installing the `VoyagerSecurityManager`, similar to the `AppletSecurityManager`. It distinguishes between trusted and untrusted objects/agents; these are called native and foreign objects/agents respectively.

The `VoyagerSecurityManager` is in some cases less restrictive than the `AppletSecurityManager` (e.g. network access is allowed to foreign objects) and in other cases more restrictive (e.g. foreign objects cannot manipulate threads/thread groups) . A complete

list of restrictions can be found in [Obje97]. However, the programmer is allowed to implement and install its own security manager.

Integrating agent systems

Voyager provides a standalone agent system called the Voyager server. Embedding an agent system in an application is as simple as calling the static method `Voyager.startup()` with a TCP port number to listen to parameter. An application with embedded agent system functionality is called a Voyager-enabled program.

Applets can also be Voyager-enabled. Because of the access restrictions an Applet is subject to, a Voyager program that acts as a software router (see Section 3) is required to run on the Web server providing the Applet. The Voyager server on the Web server needs write permissions to the Applets code base, in order to provide the Applet with the class files of objects/agents that have been loaded from another Voyager program. Using this mechanism, a Voyager-enabled Applet can not only create and send out agents/objects but also receive agents/objects. Hence, Voyager provides Applet-to-application as well as Applet-to-Applet connectivity.

Administration

Voyager does not provide special tools to administer of a Voyager-enabled network. However, the distributed delegation event model is well suited to monitor the system throughput and agent movement patterns. This can be useful for implementation of security concepts, performance optimizations, agent/object tracking tools, and for the creation of object and agent audit trails.

Usability

Like the other investigated platforms, Voyager relies on JDK 1.1 or higher. For installation, a setup utility is provided for Microsoft Windows 95/NT, and a `tar` file for other systems.

A detailed users guide (about 300 pages) covers all concepts and shows a lot of examples. For the examples source code and class files are included. Among all the platforms we investigated, Voyager is the only available as a production release (version 1.0.1) to date.

5.4 Overview of the discussed Java-based mobile agent platforms

Table 5.1 below gives an overview of the discussed Java-based mobile agent platforms and helps to compare different features.

Agent platform	Aglets	Odyssey	Voyager
Manufacturer	IBM	General Magic	ObjectSpace
Platform-specific terminology			
• Agent system	Aglet server, e.g. Tahiti, Fiji	Agent system / Engine	Voyager program
• Place	Aglet context	Place	Functionality is integrated in the Voyager program
• Agent	Aglet	Agent, Worker	Agent
• Identifier	AgletID	ProcessName	GUID

Agent platform	Aglets	Odyssey	Voyager
Manufacturer	IBM	General Magic	ObjectSpace
Communication			
• Messages			
• synchronous	Yes	N/A	Yes
• asynchronous (future)	Yes, also as HTTP requests	N/A	Yes
• oneway	N/A	N/A	Yes
• multicast	local asynchronous	N/A	oneway, publisher/subscriber
• Priorities	user configurable	N/A	N/A
• Events	delegation event model	N/A	delegation event model
• Direct method invocation	forbidden	during a meeting	allowed
Directory service	Registry/Registrant	Finder for places	Federated directory service
Persistence	user created checkpoints	N/A	user created checkpoints, automatic before/after move
Transport	ATP, HTTP tunnel	RMI, DCOM, IIOP, interface for other mechanisms	proprietary protocol on top of TCP
Security	policies configurable for trusted/untrusted agents (file system, network, JDBC, RMI client, RMI server)	Not tested	trusted/untrusted, extendible security manager
Integrating agent system			
• standalone	Tahiti	Engine	Voyager program
• embedded			
• in application	Aglet Server Development API	Not tested	Voyager-enabled program
• in Applet	Fiji, without agent receiving	Not tested (OdysseyApplet)	Yes
Administration			
• local	GUI (Tahiti), command line	N/A	N/A
• remote	HTTP requests	N/A	N/A

Agent platform	Aglets	Odyssey	Voyager
Manufacturer	IBM	General Magic	ObjectSpace
Usability			
Requirements	JDK 1.1 or JDK 1.0.2 + RMI	JDK 1.1	JDK 1.1
Installation	simple	simple	simple
Documentation	• Installation and getting started guide • Unfinished users guide • A set of example Aglets covering most of the concepts • javadoc	• Readme files for installation • Unfinished introduction chapter of the users guide • javadoc	• Users guide (about 300 pages) with detailed descriptions and examples that cover all concepts • javadoc

Table 5.1: Overview of discussed Java-based mobile agent platforms

6 Conclusion

This technology review investigated recent Java-based mobile agent platforms. Components have been defined that are common to most mobile agent platforms to date, e. g. agent system, place and agent identifier. In a separate section, features which we believe Java-based mobile agent platforms should support have been discussed. These features are agent transport, communication, persistence, directory service, security, implementation of the agent system, administration and usability requirements.

We found that General Magic's Odyssey is a simple mobile agent platform with a minimal set of functions. Also documentation is very limited. Therefore we cannot recommend Odyssey at this stage.

Both Voyager and Aglets provide a broad range of features. Voyager is already available as a production release. Although current version number of the Aglets Workbench is Alpha 5c, it is already a sophisticated platform. Voyager can be recommended as a middleware for development of large distributed applications, e.g., as CORBA replacement with object mobility. Aglets are the better choice if services are to be provided to agents implemented by many different parties, especially with the security model presented in [KLO97].

There is currently a lot of standardization activity for agents. IBM, ObjectSpace, and others, submitted a proposal for a Mobile Agent Facility [OMG97] within CORBA to the Object Management Group (OMG). This standardization effort aims for example for interoperability between different mobile agent platforms.

Acknowledgments

I would like to thank all colleagues at IBM and EPFL who have given valuable comments on early drafts of this review.

References

[BHSR97] J. Baumann, F. Hohl, M.Straßer, K. Rothermel: Mole - Concepts of a Mobile Agent System; Technical Report TR-1997-15, University of Stuttgart, IPVR.

[BFN96] T. Berners-Lee, R. Fielding, H. Nielsen: Hypertext Transfer Protocol -- HTTP/1.0; RFC 1945; 1996

[Card95] Luca Cardelli: A language with distributed scope; 1995; http://www.luca.demon.co.uk/Papers.html

[CGH+95] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris und G. Tsudik. Itinerant Agents for Mobile Computing. IEEE Personal Communications, October 1995, Page 34--48. http://www.cs.umbc.edu/kqml/papers/itinerent.ps

[Gene95] General Magic: The Telescript Language Reference; 1995 http://science.gmu.edu/~mchacko/Telescript/docs/telescript.html

[Gene97] General Magic: Introduction to the Odyssey API, http://www.genmagic.com/agents/odysseyIntro.ps

[Gray95] Robert S. Gray: Agent Tcl: A transportable agent system. In Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95), Baltimore, Maryland, December 1995.

[Gray96] R. S. Gray: Agent Tcl: A flexible and secure mobile-agent system. In Mark Diekhans and Mark Roseman, editors, Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96), Monterey, California, July 1996. http://www.cs.dartmouth.edu/~agent/papers/tcl96.ps.Z

[Java97] JavaSoft home page: http://www.javasoft.com

[Java97a] JavaSoft: Frequently Asked Questions - Java Security; 1997; http://www.javasoft.com/sfaq/index.html

[KLO97] G. Karjoth, D. B. Lange, M. Oshima: A Security Model for Aglets; http://www.computer.org/internet/ic1997/w4068abs.htm

[LaAr97] Danny B. Lange and Yariv Aridor: Agent Transfer Protocol -- ATP/0.1; http://www.trl.ibm.co.jp/aglets/atp/atp.htm

[Lang97] Danny B. Lange: Java Aglet Application Programming Interface (J-AAPI) White Paper - Draft 2; http://www.trl.ibm.co.jp/aglets/JAAPI-whitepaper.html

[LeOu95] J. Levy, J. Ousterhout: Safe Tcl: A Toolbox for Constructing Electronic Meeting Places. In Proceedings of the First USENIX Workshop on Electronic Commerce, July 1995; http://www.usenix.org/publications/library/proceedings/ec95/levy.html

[LOKK97] D.B. Lange, M. Oshima, G. Karjoth, and K. Kosaka: Aglets: Programming Mobile Agents in Java; in T. Masuda and Y. Masunaga and M.Tsukamoto (Eds.): Proc. of Worldwide Computing and Its Applications (WWCA97); Aglets home page: http://www.trl.ibm.co.jp/aglets/

[Micr97] Microsoft: ODBC 3.0 Programmer's Reference; 1997; http://www.microsoft.com/odbc/techmat.htm

[MEIT97] Mitsubishi Electric ITA, Horizon Systems Laboratory: Mobile Agent Computing, A White Paper; 1997
http://www.meitca.com/HSL/Projects/Concordia/MobileAgentsWhitePaper.html

[Mole97] Mole: Announcement of version 2.0
ftp://ftp.informatik.uni-stuttgart.de/pub/Mole/Announce-2.0.txt

[Obje97] ObjectSpace: Voyager Core Technology Users Guide Version 1.0.0;
http://www.objectspace.com/voyager/Voyager.PDF

[Odyssey] Odyssey home page: http://www.genmagic.com/agents/odyssey.html

[OMG97] Object Management Group (OMG): Data Interchange Facility and Mobile Agent Facility RFP; 1997; http://www.omg.org/library/schedule/CF_RFP3.htm

[PeSt97] H. Peine and T. Stolpmann: The Architecture of the Ara Platform for Mobile Agents; 1997; in Kurt Rothermel, Radu Popescu-Zeletin (Eds.): Proc. of the First International Workshop on Mobile Agents MA'97 (Berlin, Germany), April 7-8th. Lecture Notes in Computer Science No. 1219, Springer Verlag, Ara home page:
http://www.uni-kl.de/AG-Nehmer/Projekte/Ara/

[SFDC90] M. Schoffstall, M. Fedor, J. Davin, J. Case: A Simple Network Management Protocol (SNMP); RFC 1157; 05/10/1990.

[SBH96] Markus Straßer, Joachim Baumann, and Fritz Hohl: Mole - A Java based Mobile Agent System, an accepted paper for the ECOOP '96 Workshop on Mobile Object Systems. Mole home page:
http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html

[Sun97] Sun Microsystems, Inc.: JavaBeans 1.01 specification; 1997.
http://java.sun.com/beans/docs/index.html

[TaVa96] J. Tardo, L.Valente: Mobile agent security and Telescript. In Proceedings of the 41st International Conference of the IEEE Computer Society (CompCon '96), February 1996; http://www.genmagic.com/Telescript/Compcon96.ps

[Voyager] Voyager home page: http://www.objectspace.com/voyager/

[WPW+] D. Wong, N.Paciorek, T.Walsh, J. DiCelie, M. Young, B. Peet: Concordia: An Infrastructure for Collaborating Mobile Agents; in Kurt Rothermel, Radu Popescu-Zeletin (Eds.): Proc. of the First International Workshop on Mobile Agents MA'97 (Berlin, Germany), April 7-8th. Lecture Notes in Computer Science No. 1219, Springer Verlag. Concordia home page:
http://www.meitca.com/HSL/Projects/Concordia/Welcome.html

[YaDu96] Zhonghua Yang, Keith Duddy: CORBA: A Platform for Distributed Object Computing; Operating Systems Review 30(2): 4-31 (1996)