An Overview of Mobile Agent Systems

Hervé Paulino

Technical Report Series: DCC-02-1

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+22+6078830 – Fax: +351+22+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

An Overview of Mobile Agent Systems *

Hervé Paulino
Departamento de Informática,
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
email: herve@di.fct.unl.pt

February, 2002

1 Introduction

This paper focus almost entirely on mobile agent systems. Several systems are overviewed and some principles on what a mobile agent system should implement are identified. For each system overviewed, we focus on its execution and architectural concepts and on its implementation. The first section motivates the rest of the work; it describes why mobile agents are useful, their advantages and disadvantages, as well as some of their applications. The following sections describe agent systems, accordingly to three categories: weak migration mobile agent systems, strong migration mobile agent systems, and other relevant agent systems. The last category includes systems that do not emphasize their research work on mobile agents, but do have some characteristics that make them relevant.

2 Mobile Agents

An agent is usually defined as an independent software program that runs on behalf of a network user. It can be characterized as having more or less intelligence and it has the ability to learn. Mobile agents add to regular agents the capability of traveling to multiple locations in the network, by saving their state and restoring it in the new host. As they travel, they work on behalf of the user, such as collecting information or delivering requests. This mobility greatly enhances the productivity of each computing element in the network and creates a powerful computing environment. Mobile agents require a software infrastructure that provides them security and data protection. This infrastructure includes protocols, rules for safe mobility and, directions and directories with information about all available hosts.

Mobile code and mobile agents induce a programming model that is alternative to the traditional programming techniques and models. The reason for this is not to gain in performance, or to make new applications possible but because it supplies a general framework in which distributed information-oriented applications can be implemented efficiently and easily.

*this work was partially supported by project MIMO (contract POSI/CHS/39789/2001)

2.1 Advantages

Mobile agents introduce some advantages to the usual programming models:

- the construction of economic high quality and high performance applications. Applications using mobile agents use the network transparently, taking full advantage of the local resources. The data processing is done at the source rather than remotely fetched.

- the efficient and economic use of low-bandwidth, high-latency, error prone communications. The agent network uses a store and forward mechanism to transfer agents between nodes. This mechanism is enhanced with queuing and persistent checkpoints allowing mobile agents to use communications of mobile devices without degradation in reliability and response.

- the use of low-cost portable personal communication devices. Mobile devices are one of the areas of growth in the computer industry. Everything from laptops to palmtops, cellular phones, electronic books will access the Internet services to deliver user tasks. Nonetheless these devices will have unreliable low-bandwidth, high-latency network connections provided by telephone lines or even wireless. Mobile agents will be the ideal to accomplish the tasks performed by these devices.

- the enhancement of secure communications on public networks. As they travel, mobile agents carry with them user credentials. These are authenticated during execution in every point in the network. To enhance security, agents and their data travel along the network fully encrypted.

- the alternative to the client/server paradigm. With mobile agents the request/response architecture of the client/server paradigm is avoided, because the flow of control moves across the network. Every node becomes a server and the agent moves from location to location to find the services it needs at each point of its execution. The complicated issue of exponential scaling of servers and connections required between multiple servers becomes a simple capability issue. The relationship between servers and users is now coded in each agent, instead of being scattered along clients and servers. There is also the advantage of the hold time for connections being reduced to the time required to move the agent in or out of the host. As the agent carries its own credentials, the connection is only a move operation, abolishing authentication and spoofing. The agent moves only once, no requests flow across the connection. This allows optimization at several levels.

- software distribution on demand. The wide use of code on demand systems such as Java Applets, Java servlets and Active X has opened the door to an installation alternative: software distribution on demand, which can transport code and install packages automatically. This kind of distribution is a potential advantage for every mobile code system, therefore to mobile agents.

- asynchronous tasks. Agents can be used to perform a set of asynchronous tasks in the network. The client part of the application can be transferred from the device to the network. Once the transfer is done, the device can be disconnected and reconnected when the result is available. This is very useful in mobile devices where it is almost impossible to maintain a connection for days, weeks or longer. It is important that the agent's system is fault-tolerant, carrying his work independently of communication and host failures. It must also be aware of the exactly-once semantic, not performing a task more than once, even if it was interrupted by a failure. Some kind of check-pointing must be maintained.

2.2 Disadvantages

Nonetheless mobile agents are not the perfect paradigm and some issues cannot be overlooked. Next some technical and philosophical issues are summarized:

- the existing systems save network latency and bandwidth at the expense of higher loads on the service machines. The problem arises because most of the agents are written in relatively slow interpreted languages (most in Java), for portability and security reasons. The time saved avoiding intermediate network traffic is less than the time penalties from the slower execution and the migration overhead. Research groups have concentrated efforts in the compilation field achieving significant progress with new techniques such as just-on-time compilation and software fault isolation, that execute nearly as fast as native compiled code. Other research groups are working in reducing the migration overhead. With all these techniques available the accepting and execution of a mobile agent will be must faster than it is now, and nearly as fast as a native compiled procedure.

- almost all the mobile agent's systems allow an agent to freely move among heterogeneous machines. The code is compiled to a platform independent code (usually Java bytecode) that upon its arrival is interpreted or compiled to native code. However, for mobile agents to be widely used the code must be portable across mobile agent's systems. This will require a huge standardization effort, the OMG MASIF standard being the first step. The trend points to a standard execution environment and a standard encoding format for the agent's code and state.

- today it is possible to implement a mobile agent that protects a machine from malicious agents. Nonetheless some other security issues remain as protecting the machines without artificially limiting agent access rights; protecting an agent from malicious machines; and protecting groups of machines that are not under single administrative control. The solution to these problems must not limit the use of mobile agents in a truly open environment such as the Internet. Some research groups are working in this field and it is probable that in a few years mobile agents will be completely secure.

- the mobile agent's paradigm does not have a killer application. It can be used to implement applications which lead to faster performances in several cases, but these applications can be implemented just as cleanly and efficiently with traditional techniques. Although different techniques must be applied for different applications. So, the advantages of the mobile agent paradigm are not many when an application is considered alone. The mobile agent community must present a set of applications and argue that the entire set can be implemented with less effort and less programming groups, using mobile agents.

- it is unreasonable that any Internet service will be willing to change radically from the client/server paradigm to the mobile agent paradigm. An evolutionary path from current systems to mobile agent based systems must be provided.
- the problem of advertising must not be ignored as well. Many web sites earn money only from the advertisements. If these sites allow free access to their information to mobile agents, the number of persons visiting the site will presumably decrease.

2.3 Applications

Mobile agents have many applications, such as:

- gathering high quality information on the Internet.
- databases search. Looking for the cheapest airplane flight to Lisbon and cheapest four star hotel.
- transactions. Book the flight and the hotel. But only book the hotel after booking the flight.
- diagnostic agent. Monitor error signals and repair software when necessary on a telephone exchange. If the problem is to serious for the agent to fix, it can come back and ask the expert.
- e-commerce agent. Visit electronic shops and look for the best price, search for other e-commerce agents that have interest in an article and join forces to get a discount from the vendor.

3 Strong Migration Mobile Agents Systems

Strong migration is the highest degree of mobility. The system captures the entire agent state, i.e., data and execution state and transfers it together with the code to the new location. When the agent is received its state is automatically restored. This scheme is completely transparent, the capturing, transfer and restoration of the complete agent's state is done without any user interference. Therefore it is very attractive to users. However, this degree of transparency does not come for free. To provide execution across heterogeneous networks a global model of the agent state is required, i.e., functions to externalize and internalize agent state. Only a few languages allow these operations at such high level: Facile and Tycoon. Strong migration can be very time consuming and expensive, especially when the agent state is large and the agent is multi-threaded.

3.1 Ara

Ara [11] is a platform for mobile agents developed at the University of Kaiserslautern. Mobile agents in Ara are programmed in some interpreted language and executed in an interpreter for this language, using a special runtime system for agents called the *core*. The language specific issues must be isolated in the interpreter, while the core must concentrate in all language independent funcionalities. Ara does not prescribe an agent programming language, instead it provides an interface to attach existing languages. Since part of the agent's state is contained in its interpreter, these must be extended by state capturing functions. Currently Ara supports Tcl and C/C++, the latter though a precompilation into an efficiently interpretable bytecode. A Java interpreter is being developed and Pascal and Lisp are being considered. The core is kept to the necessary minimum, higher-level services are provided by dedicated servers. The ensemble of agents, interpreters and core runs as a single application process on top of an unmodified host operating system.

Ara agents are executed as concurrent processes, using a fast thread package, and are transparently transformed into a portable representation whenever they want to move. The system also uses operating system processes to execute trusted, not mobile, processes compiled to native code. This is used to perform certain internal tasks. Employing threads keeps the management of agents completely under the control of the core.

Ara has a special core call, *ara_go* in the Tcl interface, that allows agents to migrate at any point of their execution, using a strong migration model. Ara agents move between *places* that are virtual locations within the system. A place has a name that identifies it uniquely and serves as a destination of a migration. A place name is a list of URLs corresponding to the different transport protocols, a site and a local name of meaning to the targeted. This name will identify a place using a simple hierarchical name space. Agents have names as well, and these include a globally unique id, an identifier of their principal and an optional symbolic name from a hierarchical name space.

Agents in Ara communicate through *service points*, that are meeting points where agents located at a specific place can interact as clients and servers through an n:1 exchange of synchronous request and reply messages. In spite of emphasizing in local interaction Ara provides simple asynchronous remote messages between agents. A message will be delivered to all agents at the indicated place that are subordinates of the indicated recipient name in the sense of the hierarchical agent name space. This feature may be used to send place-wide multicast or to implement application-level transparent message forwarding.

Interpreters in Ara are the first layer of security, ensuring memory protection. Places establish a domain of logically related services under a common security policy governing all agents at that place. The central function of a place is to decide on the conditions of admission of an agent. These are expressed as an *allowance* conceded to the agent for the time of its stay. An allowance is a vector of access rights to several system resources, such as files, CPU time, memory, or disk space. may be quantitative, e.g. CPU time, or qualitative, e.g., domains where connections are allowed. An incoming agent specifies the allowance it desires and the place decides the allowance to concede. The core will ensure that an agent never oversteps its allowance. Besides local allowance an agent can be equipped with a global allowance at creation time. This allowance will limit the agent's actions during its lifetime and the core ensures that the place does not give a local allowance that exceeds the global one. Agents may form groups to share global allowances.

Authentication in Ara is based on digital signatures using public key cryptography. However, mobile agents change during their itinerary, so they cannot be signed in whole by their principal. The agent's code is wholly signed by its principal, which disables dynamically generated code, common in the Tcl language. Other security relevant components of an agent can be authenticated by dedicated schemes, e.g., the itinerary record can be incrementally signed by the nodes the agent has passed through. Agents can optionally be encrypted by public key cryptography. Ara does not supply a key distribution server, but assumes the existence of a well known trusted public key server.

To be fault tolerant agents can checkpoint at any time by recording their current internal state in a persistent media. The checkpoints can be used to restore the agent's state.

3.2 D'Agents

D'Agents [6], formerly known as Agent Tcl, is a mobile-agent system developed in the Darthmouth College in Hanover, New Hampshire. It is based on the Tcl scripting language, to run agents, and the Safe Tcl extension, to ensure security. It allows agents to migrate from host to host and access resources across the network, to create child agents, to perform subtasks and to communicate with other agents, local or remote.

The D'Agents system reduces migration to a single instruction, like the Telescript *go*, that can be inserted at any point. This instruction does not require state information from the user. All communication between agents is transparent, the transport mechanisms are hidden from the programmer. D'Agents can support multiple languages, since it allows a simple addiction of languages and transport mechanisms back-ends.

The architecture of D'Agents builds on Telescript's agent model and ARA's multiple languages. The transport mechanism is based from two predecessor systems developed at Dartmouth, e.g., TIAS. The architecture is divided in four levels. The lowest level is an API for the transport mechanisms. The second level is a server that runs in every node. The third level is an interpreter, presuming that most of the languages will be interpreted, due to portability issues. The forth and upper level consists on agents.

The second level's server is responsible for:

- status. Keeps track of the agents running in his host and answers queries about their status.
- migration. Receives any incoming agent, authenticates the identity of its owner, and gives it to the appropriate interpreter. The server also selects the best transport mechanism for transferring agents.
- communication. Allows agents to communicate with each other. It supports *event messages* that provide asynchronous notification of an important occurrence and *connection messages* that request or reject a direct connection. A direct connection is a named message stream between agents.
- nonvolatile store. Provides access to a nonvolatile store so agents can be restored in case of a machine failure.

Each interpreter in the third level has four components:

- the **Interpreter** itself that interprets the agent's code.
- the **Security Module** that prevents an agent from taking malicious action. It does not determine access restrictions. Instead it ensures that agents do not bypass the resource managers nor violate the restrictions imposed by them.

- the **State Module** that captures and restores the executing agent's internal state. It provides two functions: *captureState* that constructs a machine independent byte code that represents the agent's internal state, and; *restoreState* that restores the internal state from the machine independent byte code.
- the **Server API** that handles migration, communication and checkpoint. It is a language specific wrapper to the second level's generic API.

Adding a new language to D'Agents implies writing these four modules, if the interpreter is not already available.

The implementation of D'Agents has two major components: the server and a modified version of Tcl plus a Tcl extension. The server accepts, authenticates and starts incoming agents, buffers incoming messages, provides the flat namespace and answers the queries of the agents running on his host. It is implemented as two cooperating processes, one watches the network and the other maintains a table of running agents. In the second component, the modified version of Tcl provides the explicit task and state-capture routines. The extension provides the commands needed to migrate, communicate and create child agents. An agent is simply a script that uses these commands, running on the modified version of Tcl. A brief description of the more important commands follows:

- **agent_begin** registers an agent with a server and obtains a name in the namespace. A name consists of the IP address of the server, a unique integer and an optional symbolic name.
- **agent_submit** accepts a script, creates a child agent on a particular host, and sends it to the destination server.
- **agent_jump** migrates an agent to a particular host.
- **agent_send / agent_receive** send and receive messages.
- **agent_meet** sends a connection request to an agent.
- **agent_accept** receives a connection and accepts or rejects it.

D'Agents currently solves two security problems: protecting the machine and protecting other agents. Currently work is being done to solve two other problems: protecting the agent and protecting a group of machines. Authentication in D'Agents is based on PGP. When an agent registers, the request is digitally signed using the owner's private, encrypted using the server's public key, and sent to the server. The server checks if the agent's owner can register on the host and if so records its identity. The IDEA private key is then used as a session key for all further communication between the server and the agent. This key is needed to prevent masquerading. If the communication is local there is no need to encrypt, since messages cannot be intercepted. Instead the session key is only included in the message, so the server can compare it to his recorded session key. To prevent replay attacks a sequence number is added to every message. When an agent migrates it is digitally signed with the current server's private key and encrypted with the destination server's public key. The present implementation has two weak points. There is no automatic distribution of the PGP public keys, which implies that all servers must know the possible keys in advance. And, the system is vulnerable to replay of messages between servers. A solution is for every server to have a distinct sequence number for each server it is in contact with.

Agents have access restrictions over resources. In D'Agents the resources are divided in two categories: *indirect resources* that can only be accessed though another agent and *built-in resources* directly accessible through language primitives. For indirect resources, it is the agent that controls the resource that enforces the access restriction. The agent uses its own local access list and a 5-tuple attached to each incoming message by the local server. The 5-tuple contains the identity of the agent's owner, the identity of the sending server, a flag indicating whether the owner should be authenticated, a flag indicating whether the server should be authenticated and a numerical confidence number that represents how much trust the local server places in the sending server.

For built-in resources D'Agents uses Safe Tcl, linking dangerous commands to safe ones in the trusted interpreter. This is done to avoid removing all dangerous commands from the interpreter. Instead Safe Tcl examines the arguments and the identity of the script's owner to decide if the command shall or shall not be allowed to execute.

4 Weak Migration Mobile Agents Systems

The weak migration scheme is less transparent than the strong migration. Only the data state information is transferred. The state's size can be even more reduced, introducing user interference to select the variables that compose the state. The programmer is now responsible for encoding the agent's execution states in program variables. He has to provide some kind of start method that, based on the encoded information, must decide where the execution continues after the migration. This feature greatly reduces the amount of state to be transferred. On the other hand the semantics of the migration changes, which implies a more detailed programming approach.

4.1 Aglets

Aglets [10] was developed at IBM Japan and implements its mobile agents, named aglets, as Java objects. It supplies a simple and flexible programming interface known as the J-AAPI (Java Aglet Application Programming Interface), that allows agent developers to write platform independent aglets. The J-AAPI supplies methods for aglet creation, message handling in the aglet, and initialization, dispatching, retraction, deactivation/activation, cloning and disposing of the aglet. It is defined by a set of interfaces: *AgletContext*, *FutureReply* and *MessageManager*, and a set of classes: *Aglet*, *AgletIdentifier*, *AgletProxy*, *Itinerary* and *Message*.

Aglets defines a set of abstractions and behaviors needed to implement mobile agent technology. The main abstractions are:

- **Aglet**: mobile Java object that visits Aglet-enabled hosts in a computer network. Once it arrives at a host it runs on its own thread, therefore it is autonomous, and it has the ability to respond to incoming messages, therefore it is reactive.
- **Context**: the aglet's workspace, is a stationary object that provides the means to

maintain and manage running aglets in a uniform execution environment where the host system is secured from malicious aglets. A host can handle multiple contexts.

- **Proxy**: serves as a shield that protects an aglet from direct access to his public methods. It also can hide the true location of the aglet, providing location transparency.
- **Message**: object exchanged between aglets. Aglets supports synchronous and asynchronous message-passing.
- **Message Manager**: controls concurrency of incoming messages.
- **Itinerary**: the aglet's travel plan.
- **Identifier**: the aglet's identifier. Globally unique and unchangeable throughout the aglet's lifetime.

The behaviors supported by Aglets are:

- **Creation**: the new aglet is assigned an identifier, inserted in the context and initialized. It starts running after a successful initialization.
- **Cloning**: an aglet can be cloned into an almost identical copy. This copy runs in the same context and differs from the original only in the assigned identifier.
- **Dispatching**: transfers an aglet from the current context to the destination context, where it will restart its execution.
- **Retraction**: removes an aglet from the current context and inserts it into the context from which the retraction was requested.
- **Deactivation**: removes temporarily the aglet from its context and stores it in secondary storage.
- **Activation**: restores an aglet in a context.
- **Disposal**: halts aglet execution and removes it from its current context.
- **Messaging**: includes sending, receiving and handling synchronous and asynchronous messages.

Aglets classifies aglets into two categories: trusted and untrusted. The aglet's *Security Manager* checks whether the aglet is allowed to access the file system, the network, and other aglets. The decision to trust an aglet is exclusively up to the host.

4.2 Concordia

The Concordia system[14] is a full featured environment for the development and management of network mobile agents. It is a commercial software product manufactured by Mitsubishi Electric ITA. It consists on Concordia servers and agents, both written in Java. Each node running a Concordia server or agent runs the Java virtual machine. A Concordia server consists of multiple components that manage the agents' code, data and movement. It runs on every node of the network where agents need to travel, both user and server nodes. A Concordia server is aware of the other servers and connects on demand to transfer agents.

An agent transfer is initiated by the agent itself that invokes the server's methods. Its then suspended by the server that creates the image to be transferred. To know the agent's next destination, the server consults the itinerary object owned by each agent. The destination host is then contacted and the agent's image sent. On reception the agent is queued for execution, beginning its execution according to the method specified in the itinerary. The credentials of an agent are sent with it automatically and its access to the services is always under local administrative control.

Concordia agents support mobile computing, as well as off-line processing and disconnect operation. To provide an easier agent programming and use Concordia hides the details of the underlying communication from the programmer and the user. It ensures agent security by carrying in each agent, its creator's identity as well as his permissions. Agents in Concordia are reliable, every execution in an agent is preceded by check-pointing, and execution can be rolled back to any checkpoint if necessary. All the objects created by an agent are check-pointed as well. Concordia agents can collaborate to execute tasks in parallel over multiple servers or multiple networks.

The architecture of Concordia consists of multiple components, all written in Java. Now follows a brief overview of these components.

The **Agent Manager** provides the communications infrastructure that allow agents to be transferred and received by a node. It abstracts all communication aspects so that the agent programmer does not have to know network specifics. The Agent Manager also manages the agent's life cycle, providing for its creation and destruction. It also provides the execution environment.

The **Administration Manager** provides the administration of a Concordia server. It manages all the services provided, including Agent Managers, Security Managers, Event Managers, etc. The Administration Manager can be managed from a central location, so only a manager is required in a Concordia network, although more managers can be employed. The Administration Manager provides a single graphical interface to the administration of all the nodes in the Concordia network. The administration can be divided in two major tasks, managing servers and managing agents. The Administration Manager allows the following actions to be performed over servers:

- start and stop Concordia servers and managers
- upgrade and install Concordia servers and installed software
- monitor Concordia server performance
- view Concordia Server logs
- manage the Concordia persistent store
- manage the Concordia queues

The Administration Manager allows these actions over agents:

- install and remove agent code and libraries
- manage agent itineraries
- remotely launch agents
- terminate, suspend and resume agents
- monitor individual agent operations

The Administration Manager also manages Concordia security. The operations allowed are:

- management of trust relationships between Concordia servers.
- user permission administration: user account, group and access to services.
- encryption key administrator.
- monitor key administration.
- monitor security statistics.

The **Security Manager** is responsible for identifying users, authenticating their agents, protecting server resources and ensuring that agents move across systems secure and integer, as well as their data objects. The security credentials used by the Security Manager may come from several sources. In secure, self-contained systems, credentials may not be needed. In systems that include public networks, encryption may be required, but credentials may only contain user information, such as name and group id. In wide systems deployed in the Internet, strong authentication and security can be provided from external authorities such as Verisign. The Security Manager has an interface integrated in the Administration Manager, whose configuration and monitorization capabilities are described above.

The **Persistence Manager** maintains the state of agents as they travel around the network. It also can be used to checkpoint and restart agents when a system failure occurs.

The **Event Manager** is responsible for the registration, posting and notification of events to and from an agent. It can pass event notifications to an agent in any node of the network. It is through the Event Manager that agent collaboration is possible.

The **Queue Manager** handles the scheduling and the possible retries of agent movements between Concordia Systems. It also provides the mechanism to handle priorities and managing the agent's execution.

The **Director Manager** provides naming service in the Concordia network. It may consult a local name service or pass requests to other name servers.

The **Service Bridge** provides the interface from Concordia agents to the services available at the several hosts in the Concordia network. It consists in a set of programming extensions that provide access to the native API's as well as interfacing them to the Director Manager and the Security Manager.

The **Agent Tools Library** provides the classes needed to develop Concordia mobile agents.

4.3 Gypsy

Gypsy [7] is component-based mobile agent system developed at the Technical University of Vienna. A mobile agent in Gypsy is a Java *Runnable* object, therefore it can be executed as a dedicated thread at special places. It consists of its code and its persistent state. Three different types of agents are supported in Gypsy: one-hop agents, multi-hop agents and embedded agents. All them are implemented as *JavaBeans*, which permits a generic JavaBean-enabled user front-end where users can compose and launch their agents simply with the use of drag and drop facilities. It also permits the integration of new agents and components from third party providers.

One-hop and multi-hop agents have a *GypsyAgentInfo* object that contains their attributes, requirements and purpose to the server. This object includes an unique agent identifier, the version number, the manufacturer, the creation time, the owner's name, the owner's e-mail address, the URI of the agent's codebase and, in multi-hop agents, the URI of the home server of the owner. One-hop and multi-hop agents also have MessageHandlers to handle one-way and asynchronous communication between each other.

One-hop agents can only hop once from one server to another. When they arrive at the destination server, they are started and run their tasks until they are upgraded, removed or reach their deadline. These agents are mainly used by the remote administration tool to transfer new funcionalities or upgrade existing ones on the server. There are two types of one-hop agents:

- **Communicators** can communicate with each other over the network. Each agent has one or more communicators. There are three types of communicators: the *AdminComunicator* that enables the system administrator to maintain a server from a remote place, the *AgentCommunicator* that transfers agents to a remote location and the *UserCommunicator* that allows a user to query the status of a special agent and to retrieve waiting agents. The current implementation uses two kinds of agent communicators, the *RMIAgentCommunicator* that communicates using Java RMI and the *EmailAgentCommunicator* that sends serialized agents as MIME attached to mails. The data may be sent directly or encrypted using PGP. Each email communicator has a special email address where a local monitor thread accepts incoming mails.
- **Places** provide the interface to the underlying operating system services for mobile agents. The Gypsy server handles the agent to the place that runs it in the agent's own thread of control. When it finishes an agent requests the place to be transferred to the next place. Each place has a passive blackboard where agents register with their Id, their next location and their deadline. This mechanism allows special agents named Messengers to interpret the blackboard entries and follow an agent in order to meet and notify him. The blackboard clears its entries according to the agent's deadline. Gypsy provides special script places to support execution of scripts written in Java or Python. To enable detached computing the returning agents wait on a *UserPlace* at a home server until the user retrieves them. Each user must have a user place on a special home server to allow agents to return.

Multi-hop agents are the default agent model in Gypsy. They can hop between different locations on the basis of a preset itinerary or freely, searching for new locations to visit.

Multi-hop agents have an *Itinerary* object containing a list of locations to visit and a history log to keep track of the status and failures during the travel. It also has a *Result* object that stores in a hierarchical tree the results obtained. Each node of the tree contains the agent's ID, the server's URI, the place on the server and the result itself. There are two types of multi-hop agents:

- **Task-specific Agents** accomplish a special task. The functionality can be programed directly into the agent, which can be used as a stand-alone agent. In a second approach it can be derived from the abstract class *Task* which is an embedded agent. These can be added to *Supervisor* or *Worker* agents to gain mobility. A *Worker* is a generic task-specific agent that carries and runs one task with given constrains. *Workers* are necessary to decouple complex funcionalities from the mobility aspects. Each *Worker* has a *ConstrainManager* that monitors if the tasks are fulfilled. A *Worker* with its task is the default mobile agent model in Gypsy.
- **Supervisor Agents** have a collection of embedded agents with their tasks. These can be plugged into the *Supervisor* dynamically through the user interface. This mechanism allows users to choose lightweight or heavyweight implementations of special agents. Each *Supervisor* also has a *ConstrainManager* that decides when a task has been fulfilled and when it can carry out transactions based on the given constrains.

Embedded agents also have specific tasks but cannot travel of their own volition. They can be used as stationary agents on a server or be plugged into a mobile agent.

All servers are processes with their own Java virtual machine and can run two types of agents: communicators and places. Gypsy supports several types of servers:

- **Basic Servers**: a server offers one or more services to the mobile agent that enters it. Each server has an *AgentCommunicator* for transferring and receiving agents and for handling incoming/outgoing messages from/to the owner or other agents. It has also a *AdminCommunicator*, where the system administrator can manage the server from a remote administration tool. Each server has also several places, a *LogPlace* where the local log messengers move to print their entry, and one or more places for special tasks: a virtual shop, an interface to a database, etc. An incoming agent is checked by the network communicator for its constrains and then transferred to the desired place where it is executed.
- **Home Servers**: a home server provides detached computing. It is a dedicated server at a host that is always connected to the network. A home server has a user place for each local agent system user. To communicate to the user front-end home servers use the *UserCommunicator*.
- **Log Servers**: a log server is a centralized server in a region, from which all *LogMessengers* can be forwarded. The log messages are stored in a database for further evaluation.
- **User Front-end Server**: the front-end needs a server called *AgentManager* to transfer and receive user agents. The user builds its special agent with tasks and constrains that are added to the supervisor agent. The agent is transferred by the *AgentCommunicator*. If there is only one task a worker may be used instead of a supervisor.

- **Place Registry**: the place registry is a server that can be queried to find places. It stores a list of valid locations with their properties. The entries are stored in LDAP (Lightweight Directory Access Protocol) using its funcionalities, such as replication and management tools.

The remote administration tool provides the centralized maintenance of all Gypsy servers inside a region. Agents can travel between places which can be identified by locations. A *Location* consists of a unique server name, the place name and a list of communicators URIs with which the place can be reached. The server is responsible for registering and unregistering its places in the place registry.

Servers are protected from malicious agents through a special Java security manager and class loader. The server administrator can define runtime security restrictions such as file access, hosts and ports to contact, etc. He can also define a list of URIs from trusted codebases. If the agent's cannot be retrieved from the given list, the agent will be revoked. The agent communicator only accepts signed jar archives as valid codebases and stores them in the local cache.

4.4 Mole

Mole [12] is a mobile agent system developed in the University of Stuttgart. It uses Java as the implementation language of the agent system and of the agents. Mole agents are clusters of objects that only have references to objects it owns. References between agents are symbolic. Each agent has a unique name and is able to communicate with other agents through defined communication mechanisms.

The Mole system consists of a set of locations where agents can compute and communicate with each other and, a set of user resources and services from the underlying system. The resources and services are represented by system agents and can be used, by the user agents, by communicating with the system agents. A *location* is an abstract location with almost minimal communication costs regarding local inter-agent communication. One machine may contain several locations and locations may move between machines. Each location has a unique name, allowing the use of DNS services. At application level only the DNS names of the locations are used, providing full physical location transparency.

Locations are managed by the *engine*. The engine offers a *class server* and a *inter-location communication service*. The class server is responsible for getting unknown classes needed for an agent's execution on a location. The classes for all locations are stored in a cache. The inter-location communication service is responsible for communication to other agents and fast communication between the locations of the engine.

Mole uses the weak agent migration model, an approach to the strong model is planned for later versions. So, when the *migrate* statement is invoked, a stop method, that each agent can implement, is called. In this procedure an agent can minimize the data to send and prepare to migrate. Then the agent is sent to the destination location, serialized using the Java object serialization package. The destination location examines the received serialized data state and requests the classes that are not locally available. After the acknowledgment that the agent has successfully started, the sending location deletes the agent.

After the agent's creation in the destination location the same *start* method, executed when the agent is first created, is called. This method can consult the data state and decide what to do. This method of manually encoding the execution state on the data state simplifies a lot the migration implementation. However, it imposes on the programmer the task of encoding the execution's state manually. If an agent has several execution points, threads, these are not migrated, for complexity reasons. The *stop* method invoked when the agents stops, waits for the conclusion of every thread. Again the programmer must be aware of this problem.

Mole provides local communication and global communication. To provide global communication it supplies a set of low-level communication protocols. Higher-level protocols can be built on top of these or as agents offering the protocols as services. In the current implementation of Mole two forms of communication are realized:

- **RMI**. Agents can communicate by calling a public method of another agent. The method is executed concurrently in the called agent and the result returned. RMI can be used in local communications. It would be possible to call public methods directly in the same location. But to retain the uniformity of the method call, and to keep all method call under the control of the agent system, this is not done.
- **Message Passing**. In Mole messages are used to transfer data between agents. The send operation is asynchronous and returns immediately. It has as parameters the sender and the receiver address and the message contents. If the receiver exists at the destination location, the message is delivered automatically by calling a special method which has to be user implemented. This method is always executed in a local thread. If the receiver does not exist, the message is queued for some time and then, if not delivered, sent back. If the destination location cannot be reached for a configurable time, the sender is informed.

The communication mechanisms in Mole only supply physical location transparency. Therefore, when communicating to an agent it is necessary to known its current location. In later implementations full communication transparency will be added.

4.5 X-Klaim / Klava

X-Klaim (eXtended-Klaim) [13] is an experimental programming language, inspired in the Linda paradigm, to write mobile agents and their interactions. Klaim stands for (Kernel Language for Agents Interaction and Mobility) and is a kernel language that extends another kernel language, Linda. Klaim extends Linda by handling multiple distributed tuple spaces, which are accessible through their locality. In X-Klaim it is possible to program distributed systems composed of several components interacting through multiple tuple spaces.

The distributed tuple spaces are contained in hosts. Each host contains a tuple space and executing processes, and it can be accessed through its locality. X-Klaim distinguishes two types of localities: *physical* and *logical*, and reserves a logical locality, *self*, through which processes can refer to their execution host. The translation of logical localities to physical ones is done through *Environments*, partial functions from the logical localities set into the physical localities set. Every host has an environment, where *self* maps onto the physical locality of the host. Processes may also have Environments that have precedence over the one from the host.

Tuples are containers of information items, that can be expressions, values, localities, processes and variables. A tuple space is a collection of tuples. To select tuples from a tuple space X-Klaim uses pattern-matching.

Processes can be executed concurrently at the same site or at different sites. X-Klaim provides a high-level syntax for processes, it includes: variable declarations and assignments, conditional process expressions, sequential composition and iterative statements. It also provides special syntax for standard primitive types, as integers or strings. As an extension of Linda, X-Klaim provides the operations over the tuple space. A brief description follows:

- **in(t)@l**: evaluates the tuple *t*, looks for matching tuple *t1* located at *l*. When found the tuple is removed from the tuple space. If no matching is found the operation is suspended until one is available.
- **read(t)@l**: read differs from **in** because when found the tuple is not removed from the tuple space.
- **out(t)@l**: adds the tuple resulting from the evaluation of *t* in the tuple space located at *l*.
- **eval(P)@l**: spawns a process with code *P* at the host *l*.
- **newloc(u)**: creates a new host that can be accessed by the locality *u*.

X-Klaim allows timeout specification for actions. The programmer may indicate the time he is willing to wait for the conclusion of an operation on the tuple space.

Klava (Klaim in Java) [1] is a Java package that contains the classes that implement the Klaim operations. Programs can be implemented in X-Klaim and compiled for Java with the *xklaim* compiler or implemented directly in Java, using the Klava package. The Klava package supplies the following classes to provide run time support for X-Klaim:

- **Tuple**: handles tuple's funcionalities, creation, adding and retrieving of elements.
- **TupleSpace**: accesses tuple spaces and collects tuples.
- **Net**: implements the server that manages a Klaim net. A Klaim net keeps track of physical localities, nodes must login into a net server. *Net* is a multi-threaded server with a thread to manage each node.
- **NodeMessage**: implements messages exchanged in Klava.
- **Node**: implements the generic client of the Klaim net.
- **Locality**: is the abstract base class for *LogicalLocality* and *PhysicalLocality* classes. This class implements the *TupleItem* interface so localities can be used as tuple items.
- **KlavaProcess**: abstract class that must be derived to create a process. The only thing that must be done is to give an implementation to the *execute* method.
- **KlavaSecurityManager**: avoids remote process damage to the hosts. It prevents remote processes to access the local file system and from executing system calls.

The classes of the Klava package such as *TupleSpace* can be extended and customized to change some output behavior, so that graphical information can be displayed on runtime.

5 Other Relevant Agent Systems

In this section we described some relevant systems that do not emphasize mobile agents. There are a lot of agent systems such as JAFMAS [3], Bond [2], Decaf [4], etc. All developed in Java. We are going to focus on JatLite for being based on the first agent system developed in Java, Jat, and because it uses the standard KQML language (Knowledge Query and Manipulation Language) [9].

We are also going to focus on Voyager for being a very successful commercial product that uses CORBA, RMI and DCOM, providing universal communication between any program despite the model used.

5.1 JatLite

JatLite (Java Agent Template, Lite) [8] is a package of Java classes and programs, developed at the Stanford Center for Design Research, that allow users to create agent based software. It uses for communication among agents the standard agent communication language KQML.

JatLite is based on the agent theory of *Typed-Message Agents*, where an agent is defined in terms of agent communities that perform a distributed computation using typed messages. KQML is one example of such a message protocol that has a defined semantics prior to runtime. The JatLite system does not make any commitment or restriction on mobility. It allows agents, specially applets, to change their IP number at runtime, but does not provide any docking framework that allow agents to migrate to another host.

To simplify communication programming, JatLite provides a set of C++-like front-end templates for communication. There are templates for sending and receiving messages, files, emails, etc. This approach hides the underlying details of system programming or networking.

The features JatLite presents are:

- modular construction. JatLite architecture consists in layers that can be exchanged without affecting the functionality of the package.
- low-level communications based on TCP/IP.
- agent messages based of the KQML language and protocol, with built-in parsing for the outer layer of messages.
- multi-threaded operation, with multiple server sockets and message receiver sockets.
- a message router for agent registration, connection, name and password services, as well as storage and queuing of messages for mobile and sporadic agents.
- support for stand-alone agents in Java and C++, and applet agents through WWW browsers.
- built-in FTP file transfer and SMTP mail sending capabilities for stand-alone and Java applet agents.

The JatLite architecture is organized as a hierarchy of increasingly specialized layers. This approach allows developers to build their system on top of the selected layer. A description of the layers follows:

- **Abstract Layer**. Provides the abstract classes necessary for JatLite implementation.
- **Base Layer**. Provides basic communication based on TCP/IP .
- **KQML Layer**. Provides for storage and parsing of KQML messages.
- **Router Layer**. Provides name registration and message routing and queuing for agents. An agent sends and receives messages through the *Agent Message Router (AMR)*, which forwards them to their destination. When an agent intentionally disconnects or a failure occurs, the AMR stores incoming messages until the agent recovers. The AMR is very important to applet agents, since they can only initiate socket connections with the host that spawned them, due to WWW and Java restrictions.
- **Protocol Layer**. Provides several Internet protocols, such as FTP and SMTP. Built-in FTP is a convenient way for transferring large amounts of data. A KQML message will contain the file location, so the receiver can retrieve it. Built-in SMTP can send KQML as an email to an SMTP mail server, to, for example, bypass a firewall that presents socket connections.

Any layer can be extended to provide new funcionalities, e.g., the Abstract Layer uses TCP/IP but it can be extended to use UDP/IP as well.

The most important service of JatLite is the AMR. It allows agents to fail and recover, to migrate, and to be applet-based. The AMR buffers and forwards messages like an email server. Each agent makes a single socket connection to the AMR. They don't need to know the location of any agent. The messages are sent to the AMR that forwards them to the receiver. If the receiver is not connected, the message is stored and delivered when it reconnects. The messages are saved until the receiver sends a delete signal, which prevents losses of messages due to temporary agent failure.

JatLite security depends upon current open standards on encryption and authentication. The one exception is the password associated with the agent's name. This feature is necessary because JatLite allows agents to change IP addresses. When an agent disconnects or timeouts from the AMR, it may reconnect from a different IP address. Therefore, to avoid *spoofing*, it is necessary to authenticate the agent using the password that it provides. More sophisticated encrypted passwords, and possibly Kerberos and SSL, may be added to the system in the future.

KQML does not provide a connect/register standard mechanism, which makes reliable connection to foreign agents difficult. To do so, AMR implements a protocol using KQML performatives. The protocol is defined as follows:

Registering and connecting:

- First step: connect to AMR and send REGISTER message with name and password.
  - Reply: IDENTIFY message or error, if an agent with the same is registered.

- Second step: send WHOAMI message with agent's address, description, email address (optional), etc.
  - Reply: REGISTER_ACCEPTED message or error due to a unrecognizable message.
- Third step: send RECONNECT message with name and password
  - Reply: connection accepted or error if the agent is not registered or the password is invalid.

Unregistering / Disconnecting:

- Send UNREGISTER / DISCONNECT message with name and password
  - Reply: accepted or error if the password is invalid

JatLite has some limitations. For instance, the AMR distributes messages as they arrive. No ordering mechanism is provided. There is no notion of a distributed synchronized clock that allows messages to have a send time field. The AMR may become a bottleneck when communication increases, which induces a scaling problem. The messages will accumulate in the AMR's socket buffer. The AMR is also a critical point for failure, if it crashes all communication between agents will block.

5.2 Voyager

Voyager [5] a commercial product manufactured by ObjectSpace. The Voyager product line includes the ORB, the ORB Professional, the management console, security, transactions and the application server. The Voyager ORB is an object request broker that offers:

- **Universal communication**: Voyager programs may be an universal client and an universal server by supporting simultaneous bi-directional communication with other CORBA, RMI and COM programs. Voyager is also a universal gateway, and can translate messages between non-Voyager systems.
- **Universal naming service**: allows access to the many commercially available naming services through a simple API.
- **Universal directory service**: single directory that can be accessed and is shared by every client.
- **Universal messaging**: allows different types of messages such as synchronous, oneway, and futures to be sent to an object regardless of its location or model.

Voyager has been built using a layered architecture where each layer can be extended or replaced. The layers are:

- **Transport**: TCP and UDP.

- **Object Model**: CORBA, RMI, VNM, and DCOM.
- **Messaging**: sync, oneway, future, publish/subscribe, and distributed events.
- **Proxy**: dynamic and static (pgen).
- **Namespace**: CORBA, VDIR, RMI, JNDI, active directory.

The Voyager ORB offers all the features related to communication and class management. We are only going to focus on some features that are more important or innovative:

- **Distributed garbage collection**: reclaims objects where there are no more references to it. It uses an efficient "delta pinging" algorithm that keeps the network traffic to a minimum.
- **Mobility**: allows serializable objects to move between programs at runtime.
- **Autonomous Mobile Agents**: allows mobile autonomous agent creation that move between programs and continue to execute upon arrival.
- **Applets and servlets**: allows the set up of a message router and dynamic proxy in the server side, which enables the applets to communicate.
- **Multicast**: allows multicasting of messages to a distributed group of objects.
- **Timers**: common timing chores. Timer events can be distributed and multicasted.
- **Thread pooling**: used when allocating and deallocating threads, resulting in higher performance.
- **Advanced Messaging**: in addition of synchronous messages, allows the sending of one-way and future messages. One-way messages return immediately and discard the return value. Future messages immediately return a placeholder to the result, which can be polled or rad in a blocking fashion.

The Voyager ORB Professional builds on the ORB and adds significant capabilities as:

- **Graphical management console**: Voyager ORB professional supplies the *Voyager Management Console*, allowing management of Voyager servers, the monitorization of Voyager services in real time.
- **Dynamic XML**: allows an XML document with the specified DTD to be written or read in a simple Java-oriented way.
- **Universal gateway**: allows Voyager to bridge protocols between clients and services that are not written using Voyager.
- **JNDI integration**: client programs may use the Java Name and Directory Interface to access named objects.
- **Persistent replicated directory**: all local naming services benefit from this persistence.

- **CORBA naming service**: integrates CORBA naming service in the universal naming service.

- **Static proxy generation**: provides performance improvements at application deployment and enables proxy persistence and post-processing.

- **Ultra-light clients**: allows clients with 15K, ideal for creating applets that must download fast.

- **Transport connection management**: monitors the transport layer connections and allows the total number of simultaneous connections to be user-defined.

Voyager security provides secure network communication using the SSL protocol, allowing remote communication over an encrypted and authenticated channel. Voyager offers the ability to tunnel through firewalls using HTTP or standard SOCK 5 protocol.

Voyager transactions is a CORBA OTS-compliant distributed transaction facility that ensures all transactions are properly committed or rolled back. Using a two-phase commit allows multiple resources to participate in a transaction across multiple VMs. Both flat and nested transactions are supported. A *Resource Manager* is responsible for providing resources for use in specific transactions. It ensures that a resource is implicitly bound to a single transaction, avoiding resources to be registered with more than one transaction. A *Transaction Manager* ensures that transaction semantics are maintained and transaction processes are followed. A transaction is unique in its universe, therefore cooperating transaction managers must ensure that a transaction is uniquely and unambiguously identifiable.

The *Voyager Application Manager* is intended for business-application programmers and EJB (Enterprise JavaBean)-component builders. It offers a true EJB development environment that decouples application logic from systems programming logic. Build over the Voyager ORB professional it provides all its features, such as interaction with a wide range of technologies. The Voyager Application Manager provides the Voyager Management Console, load balancing, fault tolerance, database connection pooling, and transport connection pooling. It supports distributed two phase commits by virtue of being built on top of Voyager transactions.

6 Building a Mobile Agent Framework over DiTyCO

DiTyCO[16] is an extension to the process calculi based language, TyCO [17, 18]. TyCO is a concurrent object-oriented programming language defined by Vasco Vasconcelos from the University of Lisbon and implemented by Luís Lopes from the University of Porto. DiTyCO extends TyCO in a distributed context, allowing a computation to be scattered along a network and providing code mobility, essential to the mobile agent concept.

This section reviews all the relevant concepts implemented in the systems studied and brings them to the DiTyCO context. The discussion is structured in several sections: migration philosophy, architecture overview, sites, mobile agents, communication, persistent store, administration and security.

6.1 Migration Philosophy

Though aiming to be a strong migration system, the current specification, and implementation of DiTyCO uses weak migration. In a future specification and implementation, a new instruction similar to the Telescript's *go* shall be added. This instruction will enhance the properties of the system, allowing an agent to have control over its migration.

For a process to migrate between machines it must be capable of saving its execution state, or of spawning a new process whose execution state will be saved. Persistence involves the conversion of the process' state (variables, stack, and execution point) into a collection of bytes suitable for a transmission over the network. In DiTyCO this process will be transparent by sending an object through a channel.

Once received by the end point, the process must be placed in an execution queue and, as soon as possible, executed from the current execution point.

6.2 Architecture Overview

The architecture of the mobile systems used nowadays is quite standard. It consists of a layered architecture, where the layers may be extended or replaced to introduce new functionalities. The differences are mainly on the order of the layers, as illustrated in figure 1. The first approach is used in the Ara and D'Agents systems, the second in almost all of the others, for instance, in Concordia and Gypsy.

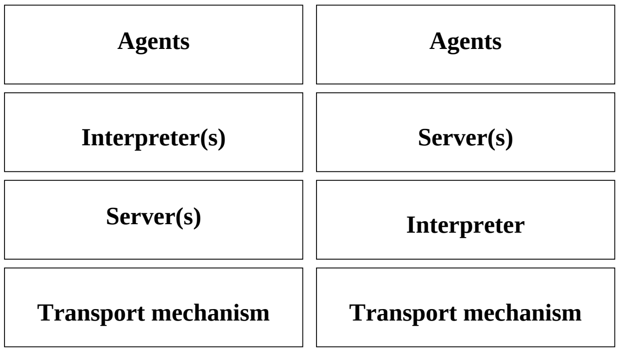

Figure 1: Different approaches to the architecture of mobile agent systems

The advantage of the first approach is the generality, allowing the use of several interpreters, i.e., several programming languages. Aiming the construction of a general framework, the choice has fallen over this approach, providing a tool as modular as possible.

The use of interpreters is required to obtain a system that can run on several machines. Compiled code runs faster that interpreted code but is machine depended, restricting a system to a set of particular machines or architectures. Ongoing investigation expects in the next years to run interpreted code nearly as fast as compiled code.

A server, as its name implies, provides a service. Therefore an interface to interact with services is needed, e.g., to build a database server that provides access to research groups' information. This interface will have to be simple and easy to use, allowing programmers to build service servers without having to know the underlying system and network details.

A special server will control incoming and outgoing agents and, the same or another server, will control the message flow. Some other tasks, such as, access restriction's control, agent's check point, will also be performed by dedicated servers.

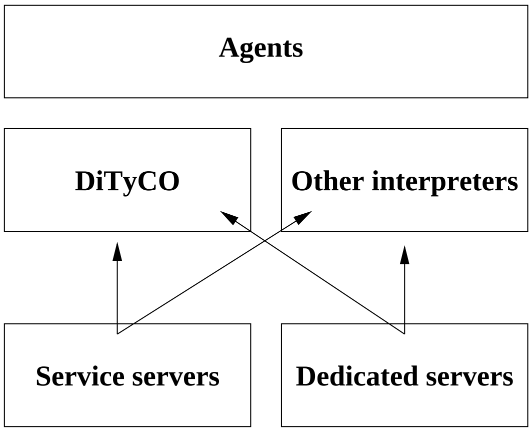

Figure 2: A mobile agent's server site

A remote management tool is fundamental, allowing server management, resource's access control, agent control, etc. Users should also have a tool to manage their agents, start, stop or retrieve, control their location, ask for their results, etc, and to be able to connect to or disconnect from the system.

6.3 Sites

The structure in figure 2 is what we will call a site. A site is the entity that hosts the servers that provide services to agents. As it is shown in figure 2 a site must contain a set of predefined features:

- **Communication**: supply inter-server communication, and manage inter-agent communication. Inter-agent communication implies an agent-name service to locate the site where the receiver agent is. The DiTyCO system solves this problem with its DiTyCO-site name server.
- **Authentication**: provides authentication of agents.
- **Security**: methods to encrypt and decrypt messages and agents' states.
- **Persistence**: supply a persistent store to agents.
- **Log**: log all the actions performed by the site.

6.4 Mobile Agents

An agent will be a DiTyCO template that must receive as arguments the agent's identification, itinerary and action, and the user's information.

```
def Agent = (id, userinfo, itinerary, action) {}
in ...
```

6.4.1 User's identification

In order to authenticate itself before a site an agent has to carry its user's information, such as login, password and access restrictions. This data structure can be implemented as an autonomous object, created from a template supplied to the user. The object must supply the following methods:

- **passwd**: changes the user's password.
- **restrictions**: changes the user's access restrictions.
- **email**: changes the user's email address.
- **home**: changes the user's home address.

6.4.2 Itinerary

An agent may have assigned to it a static itinerary, defined at creation. This itinerary contains all the sites that the agent must attend. An alternative to the static, previously defined, itinerary is a dynamic itinerary, built on the fly, e.g., a query to a search engine in order to obtain the next site where to migrate.

On both approaches the itinerary will be as a list of sites' identifications. To handle the list some methods must be provided:

- **create**: to create a new itinerary.
- **insert**: to insert a new site.
- **delete**: to delete a site on the itinerary.
- **markAsFailed**: to mark a site as failed, i.e., the agent could not reach the site.
- **getNext**: to get the next site on the list. If the next element is null, the method will search for marked-as-failed sites, to retry the migration.

The programmer defines the behavior of the agent. The itinerary may be defined as inflexible, and if so, the agent has to attend the sites in the specified order, disabling the marked-as-failed feature. If a site does not answer, the agent may interact with the user, or save its state in a persistent store and wait for the sites recovery.

6.4.3 Methods

The basic methods for the agent class are as follows:

- **disposal**: terminate execution.
- **cloning**: create new agent with the same behavior.
- **action**: the behavior of the agent.

- **saveState**: saves the internal state.

- **activation/deactivation**: restart/stop the agent's execution.

- **authenticate**: send credentials to a site.

6.5 Communication

The concept behind mobile agents is to avoid communication through message passing, migrating the code to where the receiver is and doing all the communication locally. Nonetheless, sometimes the amount of data involved in the transaction is far smaller than the size of the agent's state. So, to provide efficiency and corporate work (through agent synchronization), some message passing communication must be featured in the system.

Inter-agent communication may be divided in local and remote, and treated differently or not. Messages have to be submitted to some sort of control, to ensure reliability in communication. A message has to reach its destination or the sender must be informed. Different types of communication must be provided, synchronous, asynchronous, and maybe 1 to n communication (broadcast) as a tool of synchronization.

Several basic transport mechanisms may be offered, TCP/IP, UDP/IP, etc. Providing an easy interface for building higher-level interfaces such as KQML. Some other transport mechanisms may be included, such as HTTP or SMTP, in order to bypass firewall restrictions or to provide different ways of communicating.

Full transparent communication seems to be a good approach to inter-agent communication. The agent is referenced by its name, no need to know where the agent is to send it a message. This implies an ANS (Agent Name Service) service along with the DNS service needed for inter-server communication.

The message passing communication between agents will be achieved through the communication mechanisms available in DiTyCO: communication channels. To include some other communication features, some implementation work must be done at the DiTyCO's communication layer.

A mobile agent framework must also provide mechanisms for inter-site communication. Sites communicate to request or send an agent, or share information regarding monitoring.

6.6 Persistent Store

A site must have a server dedicated to the management of stored agents. An agent must store its state in the current site before transferring itself to a new site. This action prevents network errors: if an error occurs the state can always be retransferred. It can also be useful in security issues. If an agent is successfully attacked, the system may replace the modified agent by the saved one. To avoid replications of agents' states all over the network, the stored state may have a time-to-live or a policy that eliminates the previously saved state as soon as a new, secure one, is saved. This policy can be achieved with simple communication between sites.

6.7 Administration

The administration is necessary to manage all the services provided: agents, sites, persistence, security, etc. The administration must be done from a central location, so that only one manager is required. The administration can be divided in two major tasks, managing servers and agents. Over servers the following actions are required:

- start and stop servers;
- upgrade and install servers and upgrade installed software;
- monitor servers;
- view server logs;
- manage persistent store;
- manage queues.

The administration over agents requires these actions:

- install and remove agent code;
- manage agent itineraries;
- remotely launch agents;
- terminate, suspend and resume agents;
- monitor individual agent operations.

The administration extends itself to the security providing actions to:

- management of trust relationships between servers;
- user permission administration: user account, group and access to services;
- encryption keys administrator;
- monitor key administration;
- monitor security statistics.

6.8 Security

A mobile agent consists of an executable program that is transferred over the network. When it reaches the target host some security measures must be taken. The security issues can be divided in four major categories:

- **Protecting the host**: the hosting site must be protected from malicious agents that may consume processing cycles and memory, destroy valuable information, or bring back information useful for hacking. A standard solution is to digitally sign the agent. The digital signature will authenticate the agent and its user, before the site. The resources from the hosting site must be protected; accesses to the file system, memory, etc, must be restricted. A global access policy may be defined or agents may carry additional information that allows several levels of resource access. Agents will have to carry credentials in order to be authenticated when entering a server. A credential must include user information: the user name and password. It also may include some other relevant information, such as the user's email address, the machine where the agent was created, etc.

- **Protecting the agent**: the agent's state must be protected from modification and analysis. An encryption algorithm (such as IDEA) may be used to encrypt the whole state, avoiding malicious actions during the transfer. Nonetheless, to execute, the agent must be decrypted, which turns it vulnerable to malicious actions from the host. There is an ongoing investigation in how to protect an agent from its host. There is no ideal solution, but there are some precautions that may be taken such as confidential information that never passes through an untrusted host unencrypted.

- **Protecting a set of hosts**: an agent may not be harmful to a single machine, but consume considerably the resources of the network as a whole. For example an agent that roams though the network forever, cloning himself each time it reaches a new host. When such a situation arises an analysis must take place and the agent terminated, as well as its clones.

- **Protecting the messages between agents**: all communication must be secure; some open algorithm must encrypt all remote messages. SSL, PGP or Kerberos models are good alternatives. Mechanisms to avoid replaying and masquerading must also be included.

7 Conclusion

The state of the art in frameworks for mobile agents resort, almost exclusively, to Java as the agent programming language. The most representative of these frameworks are Aglets, Concordia, Mole and Voyager. There are some exceptions such as: D'Agents (developed in TCL); X-Klaim (developed in a Linda based language) and Ara (that supports TCL and C/C++). Although not developed in Java, X-Klaim features a package, named Klava, providing to the user a set of Java classes that simplify the programming of X-Klaim agents. Therefore building an agent construction framework over a process calculi based language is a contribution, with the promise of provably correct implementations.

Code mobility has been a growing research area into process calculi based programming languages. Several languages such as TyCO, π-calculus, Ambients and Join have been developed and implemented. Extensions of these languages have now introduced code and computational mobility: DiTyCO, Distributed-π, Mobile Ambients and Distributed Join, for example. Being an area of such interest in actual research, building a mobile agent framework over DiTyCO proves to be a state of the art area in current computer science research.

Acknowledgements the author would like to thank Fernando Silva and Luís Lopes at the Department of Computer Science of the University of Porto for their feedback and guidance through this work.

References

[1] Bettini L.. "Klava. A Java framework for mobile code".

[2] Böloni L., Marinescu D.. "An Object-Oriented Framework for Building Collaborative Network Agents". Computer Science Department, Pardue University. 1999

[3] Chauhan D.. "JAFMAS: A java-based agent framework for multiagent systems development and implementation". Technical Report CS-91-06, ECECS Department, University of Cincinnati. 1997.

[4] Graham J., Decker K.. "Towards a Distributed Environment Agent Framework", Department of Computer and Information Sciences, University of Delaware. 1999.

[5] Glass G.. "Overview of Voyager: ObjectSpace's Product Family for State-of-the-art Distributed Computing". CTO ObjectSpace. 1999.

[6] Gray R.. "Agent TCL: A flexible and secure mobile-agent system". Department of Computer Science. Dartmouth College. 1996.

[7] Jazayeri M. and Lugmayr W.."Gypsy: A Component-based Mobile Agent System". Technical University of Vienna. 1999

[8] Jeon H., Petrie C. and Cutkosky M.. "JATLite: A Java Agent Infrastructure with Message Routing", Stanford Center for Design Research. 1999.

[9] "Specification of the KQML AgentCommunication Language". URL: http://www.cs.umbc.edu/kqml/papers. 1995.

[10] Lange D.. "Java Aglet Application Programming Interface". IBM Tokyo Research Laboratory. 1997.

[11] Peine H. and Stolpman T.."The Architecture of the Ara Platform for Mobile Agents". Department of Computer Science, University of Kaiserslautern. 1997.

[12] Straber K., Baumann J. and Hohl F.. "Mole - A Java Based Mobile Agent System". Institute for Parallel and Distributed Computer Systems, University of Stuttgart. 1997.

[13] "X-Klaim. A programming language for mobile code. An overview".

[14] "Mobile Agent Computing". Mitsubishi Electric ITA, Horizon Systems Laboratory. 1998.

[15] "Mobile-Agents: The Work Force of the Millennium". February 2000, Tryllian.

[16] L. Lopes, A. Figueira, F. Silva, V. Vasconcelos: A Concurrent Programming Environment with Support for Distributed Computations and Code Mobility, in International Conference on Cluster Computing (CLUSTER2000), Chemnitz, pp. 297-306, IEEE Computer Society Press.

[17] L. Lopes, V. Vasconcelos, F. Silva: An Abstract Machine for an Object-Calculus, Technical Report DCC-97-5, University of Porto, Portugal.

[18] L. Lopes, V. Vasconcelos, F. Silva: TyCO Abstract Machine: The Definition, Technical Report DCC-97-1, University of Porto, Portugal.