Mole - A Java Based Mobile Agent System

Markus Straßer, Joachim Baumann and Fritz Hohl

{markus.strasser | joachim.baumann | fritz.hohl}@informatik.uni-stuttgart.de
University of Stuttgart, Institute of Parallel and Distributed High-Performance Systems

Mobile agents are active, autonomous objects, which are able to move between locations in a so-called agent system, a distributed abstraction layer providing security of the underlying systems on one hand and the concepts and mechanisms for mobility and communication on the other hand. In this paper, the mobility, the communication concepts and the architecture of Mole, an agent system developed at the University of Stuttgart, are presented.

1 Introduction

A new star at the sky of the distributed object-oriented systems is the emerging field of the mobile agents. Mobile agents are active, autonomous objects or object clusters, which are able to move between locations in a so-called agent system, a distributed abstraction layer providing security of the underlying systems on one hand and the concepts and mechanisms for mobility and communication on the other hand.

Although there's much ongoing discussion about what mobile agents and agent systems are, and although many claim mobile agents as their primary research interest, only few agent systems (e.g. [33][34][37][57][55]) are available at this time. At the IPVR (University of Stuttgart), a first prototype of an agent system has been developed over the year 1995. This prototype has been further developed, concepts have been changed or enhanced, taking into account the experiences won with the prototype. Design and implementation of the first version, which was publicly released in 1996, are finished, and provide a sound foundation for further studies and developments.

In this paper, we will present the basic concepts and the first implementation of the agent system Mole [74], currently using Java as implementation language as well as agent development language. First we will concentrate on some general aspects of mobile code, presenting different possible approaches of implementing it. Then we discuss the underlying concepts of mobility, including a possible definition for locations. Afterwards, the concept of agent communication realized in Mole will be presented. Section 5 introduces the architecture of Mole.

2 Mobile Agents

A typical definition of the term 'mobile agent' looks like "Some kind of a program, fulfilling autonomously a task on behalf of a user, roaming the net" [26]. While the following sections will talk about the last part of this definition ("roaming the net"), in the following we will concentrate on the first part, "some kind of a program, fulfilling a task".

In general, programs can be distributed either as machine code, executable only on one system architecture, or as machine-independent code. In the latter case, the code may be either compiled at the destination or it can be interpreted. In each case it is important that the semantics of the distributed source is unambiguous, preventing different results if the code is executed on different systems.

Having in mind to build an agent system that is able to run on heterogeneous platforms, the use of native code poses the problem that the code of an agent has to be available in native code (i.e. has to be compiled) for each system platform on which this agent is executed. This limits the possible destination platforms to a restricted set of system platforms. Adding new system platforms to this set requires to compile all available agents for this systems.

On the other hand, if machine-independent code is used, other system platforms may be integrated only by porting the agent execution environment to this platform (which also has to be done in case of machine-dependent code). Therefore, in the following, only the distribution of machine-independent code is regarded.

Using machine-independent code as distribution form, one has to decide between compilation and interpretation of the program. Compiled programs run much faster than interpreted programs, but some time is needed for compilation. For a decision between compilation and interpretation, other problem areas have to be taken into account as well (e.g. the semantics of migration, see section 3.2).

A further point are security considerations. In an agent system, the underlying computer system has to be protected against malicious agents. This includes not only control over the access to system resources, but to prevent the agents from attacking other agents or the execution environment. Therefore, the execution environment must have full control over the native instructions being executed. One possibility is the interpretation or the compilation on the fly, taking control over the instructions issued. Another possibility is the verification of the code (e.g. the byte code verifier of Java). A third solution, used by Omniware from Colusa Software Inc. [19], is the insertion of safeguards around dangerous code fragments.

In the Mole system, we use Java as implementation language of the agent system, and the agents as well. Java is an object-oriented language with some security concepts, allowing to construct secure agent execution environments. To run a Java program, the source program has to be compiled into a machine independent byte code which then gets executed (interpreted) by a virtual machine, allowing a quicker execution than the direct interpretation of the source. It is this byte code format we use as the transport format for the agent code.

In Mole, agents are modeled as clusters of objects without references to the outside, i.e. an agent has only references to objects it "owns"[1]. The agent is the transitive closure over all the objects the main agent object contains a reference to. This island concept [39] has several advantages for implementing the mobility of the agents, allowing a "simple" transfer of the transitive closure without having to worry about „dangling references". References between agents are symbolic references. Each agent has a unique name (provided by the agent system) which is used to identify an agent. Agents are only able to communicate to other agents via defined communication mechanisms. This offers the ability to use different agent programming languages, converting the information transparently when needed.

3 Locations and the concept of mobility

3.1 The location concept

One of the key issues of a mobile agent is - the name hints to it- the mobility of the agent, i.e. it's ability to move actively[2] - to migrate - within the agent system, which is, in general, distributed over several hardware platforms. The ability to move presupposes something to move to, a destination. In an agent system, one can identify several possible destinations:

- An agent. Here another agent is used as the destination address of a migration, that means one agent migrates closer to the other agent. Hereby, full location transparency is granted.
- An abstract location. Abstract locations represent 'places' where agents are executed. In this case, we have to distinguish physical location transparency (the traditional location transparency, which is granted here) and abstract location transparency. In general, an abstract location may be distributed over several physical machines, one machine may contain several locations and locations may be moved between machines.
- A physical location. Physical locations also represent 'places' where agents are executed. In contrast to abstract locations, a physical location always implies the machine executing the location. Here, no location transparency is provided.

Obviously, the last solution is a very restricted one and not suitable for a modern distributed application. Examining the first two cases, one can state, that the second case is a generalization of the first one. The location represents the 'meeting place' of the migrating agent and the destination agent.

In our view, the concept of abstract locations is the more intuitive one, especially when considering meetings of several agents. Therefore, our agent system consists of

1. There is no rule without exception - every agent has a reference to its execution environment, used to execute some agent system functions.
2. not initiated by the system

an (open) set of locations. Each location represents an execution environment for agents, where agents can compute, communicate to other agents and use resources and services of the underlying system. These resources and services are represented inside the agent system by so-called system agents and can be used by other agents, the so-called user agents, by communicating with the system agents.

Each agent location has a unique name. In order to exploit existing services, DNS[1] [65] names are used as location names. At the application level, only the DNS names of the locations are used, providing (physical) location transparency. The agent system resolves these names using the DNS service[2].

One of the reasons for using *mobile* agents is the idea to minimize global, i.e. expensive, communication by replacing global communication (over a network) with migration and local communication using the mechanisms described in the next paragraphs. This and the above definition of an abstract location do not fit. Distributing a location over several machines connected by slow connections (compared to local communication) means slow communication between agents residing on this location. Therefore, we deemed it necessary to restrict the definition of a location. A location is a place with almost minimal communication costs regarding an inter-agent communication on this location. This allows us e.g. to distribute an agent system on different nodes of a MIMD machine or a workstation cluster interconnected by a high speed network. On the other hand, by defining the locations in terms of inner-location communication costs, it is possible to place several locations on one machine (e.g. for organizational purposes).

3.2 The mobility concept

Mobility may basically be implemented in two different ways, remote execution and migration. Remote execution is a mechanism to start a new agent at a specified location, whereas the migration is a mechanism to continue the current execution of an agent at a different location.

In our system, user agents are able to migrate between locations, whereas system agents, in general representing some system resources or services, are not allowed to migrate.

Remote Execution

Remote execution is a rather simple mechanism. An agent (program code and data, probably some parameters) is transferred to the desired location where it is activated and executed. This can be implemented in two ways:

---

1. Domain Name System
2. In Mole, a location name is resolved to a IP-Address/Port-Number tuple

- The program code is transferred to the destination location and there, a new agent is created and started directly.
- A new agent is created at the source location but not started. It migrates (see below) immediately to the destination location and is started there.

There are several differences between these two approaches. Most important, the creation at the source location with immediate migration to the destination needs more data to be transferred and therefore more resources (see below). Another problem are different semantics in case of an error. If the destination location doesn't exist, the remote execution fails in the first case without further consequences. In the second case, the new agent has to be destroyed.

Migration

Migration is a mechanism to continue the current execution of an agent at another location. In an agent system, the migration is initiated by the agent itself and not automatically by the system (e.g. for load balancing purposes). After the command to migrate is issued, the execution of the agent at the current location is stopped, the agent is removed from the source location and transported to the destination location. There, the execution of the agent continues.

From the technical view, an agent consists of the following states:

- the program code (program state)
- the contents of the instance variables (data state)
- execution state

For a full-fledged migration, all three states have to be transported to the destination location. Therefore, one has to decide which objects belong to the agent (in case of an object oriented agent programming language) and which code is needed for these objects. Then, the program, the data (object attributes) and the execution state of the agent has to be transformed into a transport representation.

At this point, the definition of the agents as clusters without references to the outside (see above) offers big advantages. The set of objects belonging to the agent can be determined easily, and, using this set of objects, one can decide what code is needed by the agent. Furthermore, the agent can be removed completely from the source location, because no other agent has any (physical) reference to an object the migrating agent takes with it.

A full-fledged migration is a very complex operation, because the execution state has to be transported in addition to program code and data state. Transferring the execution state requires

- a transport format. If different interpreters/compilers or heterogeneous platforms should be used, an implementation independent state representation has to be found. This is a rather complex task.
- access to the execution state. The access to the execution state depends heavily on the way how the code is executed. If the code is interpreted, the interpreter has to be designed in a way that allows the extraction of the execution state of an agent and the re-insertion into the interpreter. If the code is compiled before execution, the execution state is represented by the stack, which has to be transformed into/from the transport format.

Having these problems in mind, we decided to implement a ‘weaker form’ of remote execution in our first version of Mole, sacrificing the migration of the execution state (the migration of the execution state will be implemented in later versions). After the call of the migrate statement, a stop method is called, enabling the agent to clean up and to prepare to migrate (for example stop all other threads). Then, the agent is sent to the destination location. Therefore, the data state is sent using the Java object serialization package provided by JavaSoft (SUN).The destination location examines the received serialized data state and requests the classes, that are not locally available, from the source location. After the acknowledgment of the destination, that the agent has been started successfully, the agent is deleted at the source location. At the destination location, the agent is created using the received data. Then, a start method is called. In this start method (implemented by all agents), the agent decides, depending on the data state, what it has to do. If the migration fails, the call of the migration statement returns an error message.

This method of manually coding the execution state into the data state makes the implementation of the migration much simpler, preventing the need for migration of the ‘physical’ execution state and therefore the need to change the Java interpreter (which allows the agent system to be executed on every platform where a Java interpreter is available). On the other hand, the programmer of an agent is imposed the burden to encode the ‘logical’ execution state of the agent manually.

A complex problem not mentioned up to now is the question of how to migrate an agent if it may have several execution points at a time, for example by creating several threads. Using threads, the complexity of a migration grows - not only on the technical side. The simple question not yet finally answered is, what to do, if there are several threads active in an agent and one of these threads wants to migrate. Migrating all threads? Apart from the fact that this is impossible with the mechanisms described above, it is a semantic problem, because the other threads don’t register this migration. Migrating only the thread which wishes to migrate? This would ‘split’ the agent and in our opinion cannot be the correct semantics of a migration. In the current implementation it is supposed, that the stop method of an agent stops all other threads and doesn’t return until all other threads of the agent are stopped. This imposes another burden on the agent programmer and some research work will be invested in this problem.

4 Communication concept

One of the basic abilities of an agent is its ability to communicate to other agents. In contrast to some approaches in the area of mobile agent systems to implement only local communication between agents (replacing slow communication by a migration and fast local communication), global as well as local inter-agent communication is realized in Mole. This enables the agents to decide themselves, depending on the amount of data transferred, whether to communicate globally or to communicate locally.

High level communication between agents is a research field of its own and some general communication protocols have been proposed [18][30][80]. In Mole, we chose to realize a set of commonly used low-level communication protocols. Higher-level protocols (such as KQML) can be implemented either on top of these protocols or as agents, offering special communication protocols as a service (e.g. several agents offering a distributed tuple space).

In the first implementation of Mole, the following two means of communication are realized:

- Remote procedure call (RPC) / remote method invocation (RMI): Agents may communicate with other agents by calling a method from the other agent.
- message passing between agents

There is no difference between local and global communication in the use of this mechanisms, providing some level of access transparency. The communication partners are always addressed via their name and their actual location. Therefore, in order to communicate to an agent, its current agent location has to be known (which can't be resolved from its agent name in the current implementation). For this reason, the communication mechanisms offer only physical location transparency. Future versions of Mole might implement abstract location transparency by using only the agent name for addressing (requiring some kind of communication cost model so that the agents are able to decide if its better to migrate to the communication partner).

The use of local method invocation for local communication between agents is not supported because local method invocation requires the use of references which may cause problems in the case of the migration of an agent (old reference to an agent which already migrated away, see above).

5 The Mole system

The architecture of the Mole system consists of several layers:

Agents are executed on a location, using services provided at the location. There are two different types of agents, the system agents and the user agents. System agents are agents with access to system resources, providing controlled, secure abstractions of these resources inside the agent system. Furthermore, system agents may offer access to legacy software, using the C-interface offered by Java. This does not cause any security problems, because system agents are immobile and may be started only by the administrator of the location. User agents, having no direct access to system resources, may only communicate with other agents.

The functionality of a Java location is broken down in two parts, the engine and the locations, allowing the execution of several locations on one machine. This allows to provide e.g. specific locations for different services.

The *location* manages the agents executed at this location. It starts the received agents and offers some basic services such as migration, communication, a simple yellow page service (including information about other agents at this location, their attributes, provided services) and others.

The *engine* manages the locations executed on one system. It offers services common to the locations, e.g. a class server and inter-location communication. The class server is responsible for getting unknown classes needed for agents about to be executed on a location of the engine, implementing as a side effect a class cache for all locations of the engine. The inter-location communication service is responsible for communication to other engines and for a fast communication mechanism between locations on the same engine.

An engine, it's locations and the agents contained in these locations are executed in a single Java interpreter.

6 Conclusion and future work

A first version of Mole has been published in june 1996 and work is going on. There are a lot of important features missing in this first release - versioning, accounting, trading etc. Different student thesises provide or will provide some of these and other features, e.g. a CORBA enabled trader, a multi user dungeon (MUD) on top of the Mole system, robust queues to provide a fault tolerant communication subsystem, a graphical inspector / viewer allowing the examination of locations and agents, a better persistency and migration mechanism, an orphan detection mechanism, a facility for finding agents and others.

Several projects start or have started using Mole as a base. **AIDA** is a project sponsored by the DFG, exploring the basic design and implementation of an agent system. In this project, security issues, grouping of agents and the termination of agents are important issues. **ASAP** is a project in collaboration with the swiss Nationalfonds, using Mole as a first base for the development of a system in the field of hypermedia documents (active documents), allowing to evaluate the usability of Mole. A third project is **ATOMAS**, sponsored by Tandem which focuses on the incorporation of transaction management in the agent system and the implementation of recoverable agents, agents that can be checkpointed and rolled back if a transaction fails.

Last but not least, Mole will be used as the base of our future research work, which lies in the fields of the termination of agents, orphan detection in agent systems, and in the exploration of group semantics of agents.