MAP: Design and Implementation of a Mobile Agents Platform

Antonio Puliafito, Orazio Tomarchio, Lorenzo Vita

Istituto di Informatica e Telecomunicazioni
Università di Catania
Viale A. Doria 6, 95025 Catania - Italy
E-mail:{ap,tomarchio,lvita}@iit.unict.it

Abstract

The recent development of telecommunication networks has contributed to the success of applications such as information retrieval and electronic commerce, as well as all the services that take advantage of communication in distributed systems. In this area, the emerging technology of mobile agents aroused considerable interest. Mobile agents are applications that can move through the network for carrying out a given task on behalf of the user. In this work we present a platform (called MAP (Mobile Agents Platform)) for the development and the management of mobile agents. The language used both for developing the platform and for carrying out the agents is Java. The platform gives the user all the basic tools needed for creating some applications based on the use of agents. It enables us to create, run, suspend, resume, deactivate, reactivate local agents, to stop their execution, to make them communicate each other and migrate.

Keywords: mobile agents, distributed computing, Java, network management.

1 Introduction

During the last few years, a considerable development in telecommunication networks, which has contributed to the success of distributed systems, has been observed. Computers are no longer considered devices able to access only their own resources, and to communicate with each other only occasionally. Conversely, they are now part of a global environment where local and remote resources can be shared. The development of telecommunication networks has

therefore encouraged the development of services such as e-mail, the access to remote databases, the Web, electronic commerce, and in general all those applications that take advantage from the communication among different distributed environments.

In this area, the development of so-called *mobile agents* was particularly interesting. Mobile agents are software modules able to move through the network autonomously, in order to carry out the task that they were given by the user [CHK95, GK94, EW95]. As the word itself suggests, an agent is an entity acting on behalf of someone else. It helps a user to run a specific task, either by communicating with the user who launched it, or with other agents, or with the environment in which it is. The main aspects that contributed the success of agents are their ability to operate autonomously and intelligently, and their ability to migrate. In fact, agents are applications able to carry out the task for which they were created autonomously, moving (if necessary) from a node of the network to the other, in order to obtain the information they need. We can therefore speak of mobile agents that, if necessary, carry the state (in which they were at the time of the suspension) with them, that is the set of the values taken by some variables inside the agent itself.

The first agent systems developed were based on languages not widely used, and anyway they were almost produced in the environment of academic research. The first commercial agent system was Telescript [Whi94, Whi95] by General Magic, which developed their own language and a development environment for agents. The use of Java [Gos95, Ham96, SJRK97] favoured the design and the creation of several platforms for agents. Some of the mobile agent systems developed in Java are Aglets [LO97] by IBM, Odyssey [Mag97] by General Magic, Voyager [Obj97] by ObjectSpace, and other systems produced by university research such as Mole [SBH96], and JavaToGo [LM96]. We can refer to the work [KS97], for a comprehensive review of such platforms. The main features common to such systems include some *agent servers* that, in each host, create the environment of execution for agents. Such servers supply the basic services to agents. Agents of such systems can move from a server to the other by using several mechanisms, and carrying a part of their state with them.

In this work we present MAP (Mobile Agents Platform), a platform for the development and the management of mobile agents, which was completely developed by using Java. This platform gives the user all the basic tools needed for the creation of applications based on the use of agents. It enables us to create, run, suspend, resume, deactivate, reactivate local agents, to stop their execution, to make them communicate with each other and migrate. The use

of Java (thanks to its independence from hw and sw architectures) enabled us to develop a platform able to operate in heterogeneous environments. Besides, Java is equipped with mechanisms that facilitate the dynamic execution of parts of code that can be downloaded through the network from remote nodes [WWR97]. Agents can move from a node to the other, taking a part of their state with them. In particular, we used the mechanisms of Object Serialization [Mic97] present in the latest version of Java Development Kit (JDK 1.1).

The rest of this paper is organized as follows: in section 2 we make a comparison between the traditional techniques of distributed computing and programming paradigms based on mobile code, and we examine the advantages introduced by the latter. In section 3 we describe the architecture and some key concepts of the design of MAP; besides, we describe the management mechanisms of the system. In section 4 we present some notes concerning the implementation of the system. An application to distributed network management is presented in section 5. Finally, in section 6 we present the conclusions, together with the future working directions for the development of the platform created.

2 Mobile Code Paradigms versus Traditional Distributed Computing

Traditional distributed computing has been based on the well known client/server paradigm. The mechanism on which such paradigm is based is RPC (Remote Procedure Call). This mechanism extends the traditional procedure call, and enables a process in a computer to call a procedure in another one. A communication channel between the client application and the server process is established; through this channel the client sends a request including the parameters of the procedure called. The server, after processing, sends the results back to the client. Of course, the client and the server need "to agree" on the procedures that can be accessed in remote, on their arguments and on the returned result. In such approach, each interaction between client and server requires of two messages to be sent through the network; this means that the connection must be kept open during the whole interaction. The code of the procedure to be executed is on the machine that runs it.

An alternative to such traditional mechanisms has recently been spreading, and is based on the use of environments that give a sort of *"code mobility"*. By this term we mean the possibility to change dynamically at run-time the binding between the software components of an application and their physical location within a network of computers. Even though the research about code mobility

is not totally new [BHJL88, SG90], the possibility of applying such mechanisms to distributed environments on a wide range[CPV97, MRK96], aroused our interest.

Several levels of mobility can be considered. First of all, we make a distinction between *code mobility* and *agent mobility.* In the first paradigm we can include the mechanisms of *remote execution* and *code on demand.*

In the case of remote execution, the code is transferred to a remote node, where it is run up to the end; the results are therefore returned to the node that sent the code. The transferred information include both the code to be run and the parameters needed; the node on which the code must be run is defined by who starts the whole operation. The program on the remote node, once it has been activated, can use the same mechanism for activating other executions on several nodes; the recursive application of such model leads therefore to a tree processing structure.

In the case of code on demand, the client on which the code will be run can require a specific software module from a remote server. Java applets are a very common example of such type of technology.

However, in both cases, the code is transferred before being activated. Conversely, by agent mobility we mean the possibility of transferring a software module (agent) after starting its execution. An agent starts its execution on a machine, and then can stop it, move to another machine and continue its execution there. An agent can also move several times during its execution, unlike the mechanisms of code mobility described before, in which the program, after starting its execution on the remote site, no longer moves.

Two levels of agent mobility are distinguished in literature ([CPV97, GV97]): *strong migration* and *weak migration.* An agent in execution consists of: the code (*program state*), the contents of variables (*data state*), and the stack (*execution state*). Strong migration is the highest level of mobility; all of the three components of the state are captured and transferred to the destination machine, where this state is restored and the agent continues its execution from the exact point where it had stopped. Even though such feature is very powerful and interesting from a programmer's point of view, few systems implement a complete strong migration [Whi95, BHJL88, JvRS95]. If we work in heterogeneous environments, such as the ones where the agent systems will operate, we need to adopt a representation of the state that could be moved among the different architectures. This operation is very difficult to carry out, so the systems that implement a strong migration are carried out in a homogeneous environment that is specifically created [Whi95, BHJL88, JvRS95]. Besides, in agent systems we often deal with multi-threading languages, so this operation

might reveal much more expensive and time-consuming.

For these reasons, the most common operation in agent systems is what was called weak migration, in which the execution state is not transferred. It means that the agent, once it has reached the node of destination, will not be able to continue its execution from the point in which this had been stopped, but will start from the beginning, and will keep the value of the state variables as the one before the transfer. But in this scheme the programmer must expressly save the information needed for a correct restarting of the execution within variables which are part of the data state. By examining such variables, the agent (once it has reached the node of destination) will be able to restart the execution correctly. In an agent system, unlike a system of migration of processes where migration is imposed from outside (for example for load balancing purposes), an agent migrates on its own iniziative, and the programmer can do the operations of preparation to the migration of the agent.

This load imposed to the programmer is however balanced by the fact that the state information to be transferred is much smaller than in the case of strong migration.

The use of an agent-based approach while carrying out a distributed application, gives some advantages than a traditional solution. In this case, while developing a distributed application, the interaction among the components is generally considered not dependent on their location. In some cases, it is fixed by the programmer during the implementation phase. In distributed object systems such as CORBA [TM95], the location of the components is deliberately hidden to the user, who does not need to take care, nor can see where the service required is done. In such environments there is no distinction between the interaction of objects resident on the same host and objects on different hosts. But in some situations, we need (in the phase of design) to consider the existence of different locations and of different resources in each location. As it is reported in [WWWK94], hiding or not considering that the interaction between two software components can greatly depend on their mutual location can lead to unforeseen problems of performances or of reliability of the application itself.

For example, using a scheme based on agents can therefore be useful everytime we need to use resources strictly connected to a machine. In fact, if necessary, an agent can move to the site of another agent or where a fixed resource is resident, to do the operations required without generating any traffic in the network, by using only local communications. A typical case concerns client/server applications in which the client must retrieve some data from the server and operate complex filtering operations on such data; by moving an agent containing the procedures that deal with filtering, only the data that

actually concern the client are sent through the network, with a considerable reduction of communication costs. Besides, a permanent connection between client and server is not necessary in such scheme; the agent, once it is sent to the site of destination, can continue doing its operations and can communicate the results as soon as the client connects to the network again.

The use of mobile agents can therefore be useful in several fields of application, although none of the following applications requires the use of mobile agents: in fact, each application can be run with the existing technologies [CHK95, MRK96]. However, as we said in precedence, the use of mobile agents can contribute to build these distributed applications more simply and effectively, at the same time.

These are some of the areas in which such technology can actually give a positive contribution:

- *Information retrieval:* mobile agents can be an effective tool for retrieving information within a distributed system; in fact, an agent containing the user's query can migrate to the place(s) where the information is actually stored; here the agent can do the necessary operations of research and filtering, and give the user only the useful information [EW94];

- *Electronic commerce:* electronic commerce is an increasingly developing area in the Internet; mobile agents can help the user to research the products that meet his (her) requirements, to search for the most convenient offers, etc. [Way95, Whi94];

- *Mobile computing:* mobile agents can be an effective tool for mobile computing: in fact, users want to access network resources from any position, notwithstanding the band limits due to the present wireless technologies. Users submit their requests through an agent, which runs their request within the network, and enables the user to obtain the results in another moment (so the user does not need to remain logged in, waiting for the results). Besides, this enables us to exploit the fixed calculation resources within the network, avoiding the use of mobile devices, whose calculation power and operation autonomy are often limited [CGH+95];

- *Distributed Management:* mobile agents enable us to delegate some management functions from a central station to remote nodes, thus reducing the workload on the central station, and improving the exploitation of the available band [GYM+95, PTV97];

- *Distributed Computation:* thanks to the possibility to pilot the node on

which mobile agents are run, they are a new paradigm for parallel calculation on a distributed network of workstations [BFD96];

- *Collaborative Applications:* this is a growing area of development: in this area, mobile agents might be an effective support for sharing data and any kind of documents; they can give a flexible architecture and enable users to work by sharing various network resources [WPW+94].

3 MAP: Mobile Agents Platform

In this section we introduce the agent system MAP [1] that we developed and implemented. MAP is a platform for the development and the management of mobile agents that gives all the primitives needed for their creation, execution, communication, migration, etc.

3.1 Reference Architecture

The MAP basically consists of agents that can move to the various nodes of a network, and of servers that constitute the environment in which the agents will run. The architecture of the MAP is shown in Fig. 1, in which the constituent parts of the platform, which will be described later, are pointed out.

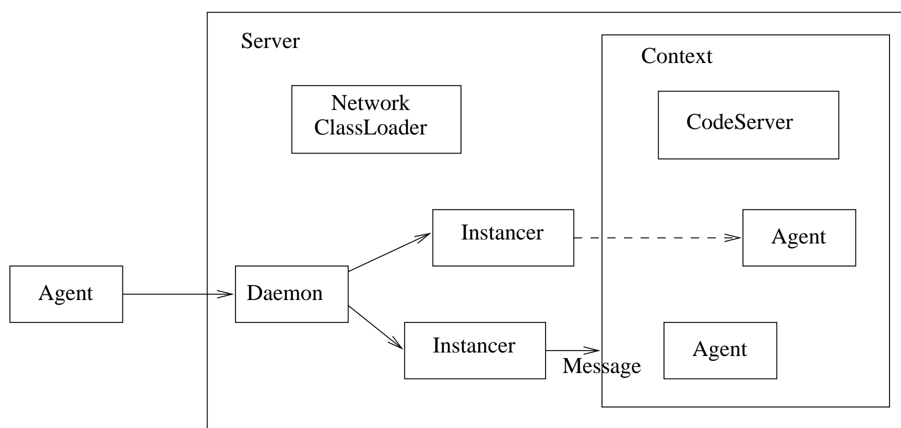

Figure 1: MAP architecture

A node belonging to the platform MAP consists of an object called Server that contains, all the entities needed for the operation of the platform itself. In a host there can be more than one Server, each identified by the DNS address of the host and by the TCP port number on which the server is listening for accepting agents or messages coming from the network.

[1] MAP is available at the following Web site: http:/sun195.iit.unict.it/MAP/map.html

Server The Server is the main object of a MAP server, in which the entities Daemon and Context and local agents are instanced. The presence of a Server on a node characterizes it as belonging to the platform MAP. Its activation enables the node to accept and have agents coming from the network run, as well as to activate other agents locally.

Daemon The Daemon is the entity of the MAP that listens on a certain port, waiting for agents coming from other nodes and for messages to be delivered to local agents. Both messages and agents travel in a serialized form. Each time a stream arrives from the network, the Daemon creates a specific entity called Instancer, whose task is to instance the serialized object, both if it is an agent and a message. In both cases, the object, once it is instanced, is passed to the Context, whose task will be to make it run (if it is an agent), or to send it to the receiver agent (if it is a message).

Context The Context is one of the basic objects in a MAP server. In fact, it knows all the agents present on the server, saved on an appropriate list, and gives the user all the functionalities needed for their management. The Context, puts some methods at disposal, which enable to create an agent, to make it run, even on a server in which its code is not present, to suspend it, to deactivate it, to resume its execution, and even to kill it, if necessary. Besides, the Context gives an agent that is running on a specific server the possibility to obtain some information about the agents that are running on the same server or on a remote server.

The Context is the element of the MAP that manages the communication among the agents. It can take place both in a synchronous and asynchronous way, and both among agents resident on the same MAP server and on different MAP servers.

Each object coming from the network in serialized form, is passed to the Context, after having been instanced by the Instancer. If such object is an agent, the Context initializes it, by giving it a reference to the Context, and later, starts its execution, and consequently updates the list of local agents. Conversely, if the object is a message, the Context makes sure to deliver it to the receiver agent, of course after checking the agent's availability to receive such message, or of its actual presence on the server.

NetworkClassLoader The NetworkClassLoader of the platform MAP is used for enabling the agents to run on a specific MAP server, even when their class is not present there, and to exchange (through messages) also objects of classes not defined locally. The Daemon, in order to use such classes transparently, once a stream of data comes from the network, creates a new Instancer object by loading it with the NetworkClassLoader. From now on, each object

to which the Instancer will refer will be automatically instanced with the same NetworkClassLoader. Thus, if one of the classes to which the agent or the message refers is not actually present locally, the NetworkClassLoader will search for it in the network, in a list of MAP servers fixed within the Context. Once the class is found, it is loaded from the remote site and saved in a cache memory managed by the Context, so that it can be accessed and used, if necessary, also by of the other agents of the server.

CodeServer The CodeServer is an internal entity of the Context, dynamically created; in fact, the Context of a platform instances a new object CodeServer each time it is requested a class by a NetworkClassLoader, either local or remote. The CodeServer is given a table of the Context in which all the classes available in the platform are saved.

As we have already described before, one of the most important features of the platform MAP is the possibility to make an agent migrate or to send messages through the network even to servers whose classes to which such objects refer are not present. In order to permit this, each time a specific object reaches a new server, the corresponding Daemon loads a new Instancer object with a NetworkClassLoader that deals with the loading of such classes. If they are not available locally, the NetworkClassLoader interrogates the CodeServer of some remote sites saved on an appropriate vector (within the Context and that can be updated dynamically), searching for the classes required. If it finds them, they are loaded from the remote site and saved in the local table of classes; from now on, they can also be accessed by all of the other agents present in the server.

3.2 Agents' Structure

As we have already said before, agents represent the entities of the platform MAP that can move in the network for carrying out a task assigned by the user who created them.

Within our platform, an agent is able to:

- suspend itself and another agent: to suspend an agent means to stop its execution temporarily, by keeping all the references to the agent active;
- resume a suspended agent: to wake up an agent means to resume its execution from the point where it had been suspended before;
- deactivate itself and deactivate another agent: to deactivate an agent means to stop its execution, by downloading the agent to a disk in a serialized form, and by deleting all the references to it;

- reactivate an agent: it means to deserialize an agent that had been deactivated before, by giving all references back to it and restarting its execution from the beginning;
- create a new agent: to create an agent means to instance a new Agent object (by setting its Context, its Identifier, etc) and to make it run;
- kill an agent: it means to stop the execution of an agent, by deleting all its references and cancelling its entry from the list of the agents in the server;
- migrate to a new server: to migrate an agent means to move it to another node, where it will start its execution from the beginning;
- communicate with other agent, through messages, both in a synchronous and in an asynchronous way.

From an implementation point of view, an agent is an object obtained by instancing a class deriving from the Agent class supplied by the platform. This enables the agent to take advantage from the methods put at our disposal by such class.

3.3 Managing the platform

The platform MAP enables the user to take advantage from a graphic interface (shown in Fig. 2) that permits an easier management of the agents in a MAP server. The graphic interface shows a window where all the agents running locally are listed, and the following information is given for each of them:

- the identifier of the agent
- the name of the class of the agent
- the current state.

If we select an agent from the list, we can do on it any operations available in the platform, through the buttons present in the upper toolbar (*suspend, resume, deactivate, activate, dispose, go*).

Conversely, by using the button *Run*, we can make an agent run (resident on a specific MAP server, not necessarily local) to a different MAP server, where the class of the agent can even be absent. Figure 3 helps us to explain this feature better. The user has activated the *MAP User Interface* on the machine `pc10a`, and can make an agent run on any node where the MAP server is active. To do this, we only need to indicate the name of the class that implements the

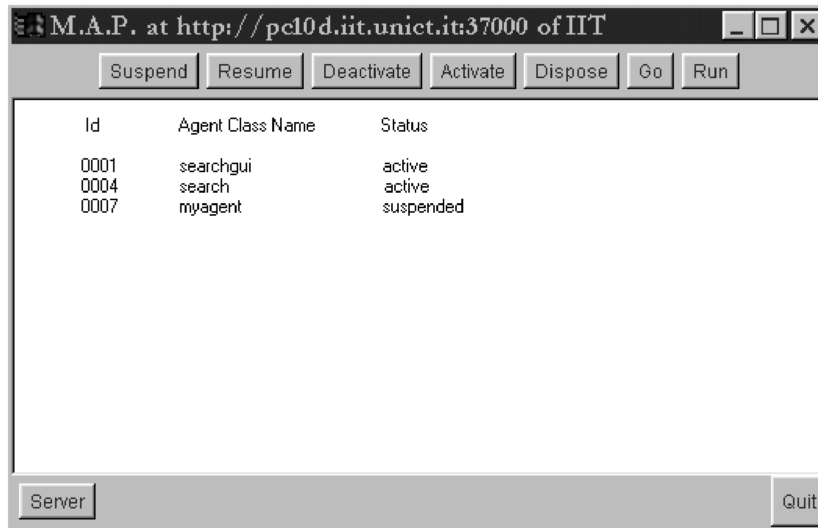

Figure 2: MAP User Interface

agent desired (in this case *MyAgent*), and the URL of the node where we want to make it run (`pc10e`). The class that implements the agent does not need to be resident in the node where there is the user (`pc10a`), or in the one where the agent will run (`pc10e`): in fact, it only needs to be in a node where the MAP server is active and that the user will have to indicate as source URL (`pc10d` in figure 3).

Thanks to this functionality, the platform presented enables us to use at best paradigms based on *remote evaluation* and *code on demand* described before.

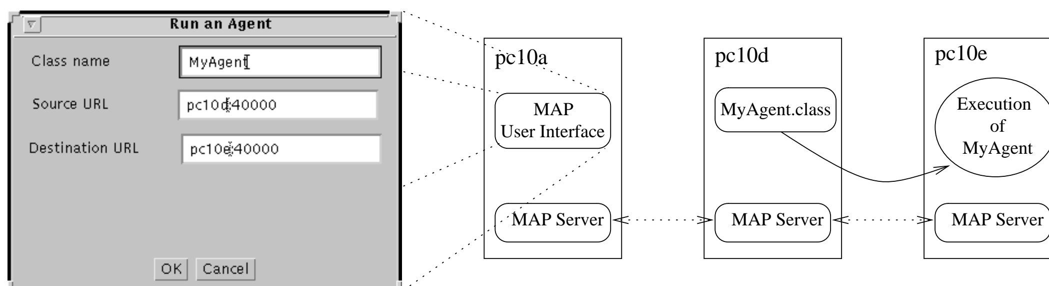

Figure 3: Running an Agent on Different Machines

The button *Server* enables us to obtain some information about the MAP servers active in the system, and about the agents in execution on such server. The window that appears (Fig. 4), is divided into three columns; the first one shows the list of the active MAP servers in the system. By selecting a server, the second column will show the agents in the selected server and their state. On the third column the classes available on the selected server are shown; they

are useful for an agent that must move to one of these nodes.

Besides, the user can add a new server to the list at any moment, through the button *Add New Server*.

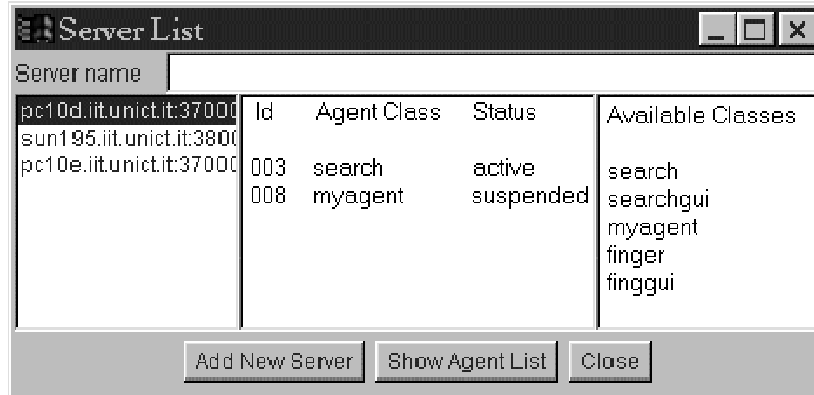

Figure 4: List of MAP Servers architecture

4 Implementation Notes

In this section we describe the main functionalities of platform MAP and some implementation details. First we describe some fundamental design issues regarding the choiche of the implementation language, the serialization mechanism for agent migration, the loading of classes from the network and communication issues among the agents. Then, we describe how the basic mechanisms provided by MAP have been implemented.

4.1 Implementation language

A fundamental feature of such a system is the possibility for agents to move and run on different architectures. For this reason, the choice of the environment where to develop such platform is very important in the design phase, and not only during the implementation phase.

The use of traditional compiled languages such as C is not convenient because they are generally machine-dependent languages, and therefore not suitable to be used in heterogeneous environments. Besides, since they are compiled languages, a considerable effort is required for carrying out a platform whose code must move from a node of the network to the other, dynamically linking new code modules. The languages that are more suitable to the development of mobile agents are interpreted and/or scripting languages, that can run on several machines, provided that the corresponding interpreter is installed in

them. However, the scripting languages belonging to this category do not give all the power and the flexibility required for developing a complete and effective platform.

For such reasons, we chose to use Java for developing the whole architecture. Java [Gos95] is an object-oriented, multithreaded language; it is portable on different hw/sw architectures, and has had a considerable success thanks to the possibility to build small applications (applets) which, integrated within Web pages, allow their execution on the client machine within the browser. But the potentialities of Java are much higher than this simple use [SJRK97, WWR97]. These are the main characteristics that, in our opinion, make it a good language for the development of a platform for mobile agents:

- it is object-oriented (and so a modular development of the code is favoured, and the user can write one's agents with limited efforts, starting from the classes provided by the platform);
- it is portable on an increasing number of hw/sw architectures;
- the presence of mechanisms for the dynamic loading of classes from different sources and for their dynamic linking at run-time in the current application;
- possibility of serializing the objects thus enabling their transfer through the network;

4.2 Agent migration

The platform implements a weak migration by relying on the mechanisms of Object Serialization present in version 1.1 of Java [Mic97]. Object Serialization is a mechanism that enables us to represent an object as a stream of bytes. All the variables of the object are stored within the serialized representation of an object, together with all the references to objects contained in it. In the previous section we pointed out the main reasons why a strong migration can be hardly implemented in a heterogeneous environment. Besides, in our case the capture, the transfer and the restoration of the execution state can cause considerable problems. In fact, Java is an interpreted language, so a part of the execution state is included within the state of the interpreter; the capture of such state becomes practically impossible without changing the interpreter. But a modification of the interpreter, as well as the intrinsic problems, would lead to the loss of portability, which (as we said before) is one of the fundamental features of an agent system.

In our platform, when an agent must migrate, it is serialized and the stream of bytes obtained is sent to the destination node through the network. Here the stream is deserialized, and the execution of the agent is restarted with the stored state.

4.3 Loading classes from the network

While serializing an object, the code of the class to which the object belongs is not stored, but only a reference to it is stored. Thus, since in our system the classes needed are not always in the arrival node, we had to take advantage from another feature of Java: the possibility to load some classes dynamically in runtime from different sources. This was done by using a NetworkClassLoader, whose task is to load the classes that are necessary for the execution of an agent. If the class needed is not present locally, the NetworkClassLoader searches for it within the nodes contained in a list specified during the configuration and the startup of the platform servers, but that can be updated in runtime, thanks to some information brought by the agents. Such mechanism permits therefore to exploit the network at best. The transfer of the bytecode of a class will occur only when required (in case of migration of several instances of the same agent, we do not need to transfer the same class); in any case, we can always do the transfer from the "closest" node, and not necessarily from the departure node of the agent.

Thanks to such mechanisms for the dynamic loading of classes present in Java, the paradigms of code on demand and remote execution that we described before have been integrated within of our platform. Such mechanisms are available for each agent, that can therefore change its behavior according to the classes that it can load on each node. The designer can therefore integrate the different paradigms of mobility that are more suitable to the specific application, even if only one working environment is used.

4.4 Communication among agents

The ability of an agent to communicate with other agents is another basic characteristics that must be given by such a system. Several communication mechanisms are possible. In the MAP the communication among the agents takes place through the exchange of messages that may be synchronous or asynchronous. In our opinion, this solution (unlike some mechanisms based on RPC) is very flexible and enables us to implement several schemes of communication and synchronization among agents. Anyway, the platform gives the basic mechanisms for the communication: any advanced schemes of communication

and co-operation (see for example KQML/KIF [FMFM94]) can be implemented beyond the primitives supplied.

The encoding of messages takes place in the same way as the migration of agents. In fact, the mechanism that we selected for transferring messages is the Object Serialization, in order to permit the sending of complex objects among different agents.

In the case of a synchronous message, we obtain a behavior similar to a RPC. In fact, the agent sender invokes an appropriate primitive, by giving the message and the identifier of the agent recipient, and stops, waiting for a reply message. Conversely, in the case of an asynchronous message, the agent sender invokes another primitive and, after sending the message, continues with its execution. Further details of such mechanism will be given later, in section 4.

4.5 Basic Mechanisms

Information about an agent
We can obtain some information about the agents instanced on a node of the system, by recalling the methods `getList` and `getAgentList`, which give us a list containing the Identifiers of all the agents in the local server and a list containing all the information concerning them, respectively. Each element of the list consists of:

1. a reference to the corresponding agent;
2. the identifier of the agent. Even though it is already contained in the agent, we needed to introduce it as another attribute for searching and finding the agent in the platform (and therefore in the list), also when the agent is deactivated. In fact, in this case the reference to the agent is set to null;
3. the name of the class of the agent;
4. a short description;
5. the name of the owner user;
6. two boolean values, *susp* and *deact*, which indicate whether an agent is currently suspended or deactivated, respectively.

Creation of an agent
The creation of an agent can be carried out with the method `runAgent` of the Context. It needs three parameters: the name of the class to be instanced, the URL (interpreted as the couple consisting of the host name and listening

port) of the MAP server where the class has to be found (Source URL) and the URL of the MAP server where the agent has to run (Destination URL). If Source URL and Destination URL are both the same as the URL of the local MAP server (Home), the class has to be searched locally (both in the directory agent and in the global cache of the Context) and run there. Conversely, when the Source URL is the same as the local URL but the Destination URL is different, the agent is instanced locally but, before it is run, it migrates to the Destination URL. Finally, if the Source URL is different than Home, a system agent, called *Mover*, is launched. It automatically migrates to the Source URL and, once it reaches its destination, recalls the method runAgent of the Local Context, whose Source URL is the same as Home and whose Destination URL is the same as the previous one. If the graphic interface has been activated, the method runAgent of the Context can be recalled by pressing the button Run. It enables us to specify (in an appropriate window) the three parameters described before.

Suspension of an agent

The platform MAP enables us to suspend the execution of an agent at any moment, and to resume it later from the point where it was suspended. An agent can be suspended by recalling the primitive `suspend`. It searches for the agent (for which we want to suspend the execution) within the list of agents in the node, and, if the primitive finds it, stops it (with the method `lock`), suspends its Thread, and later releases the agent and updates its state. We need to point out that an agent can be suspended only if it is still active, that is only if it has not already been suspended or deactivated, and if no one else is already acting on it.

Resuming an agent

The primitive `resume` enables us to wake up an agent that had been suspended, and to continue its execution from the exact point where it had been stopped. The method `resume` searches for the agent which we are interested in (within the list) and, once it has been found, stops it; then it searches and wakes up the corresponding Thread of execution and, after that, releases the agent and updates its state. Of course, we can resume an agent only if it is actually suspended, and if no one is already acting on it.

Deactivation of an agent

An agent can be deactivated with the method `deactivate`. To deactivate an agent means to download it to a disk in a serialized form and cancel the reference to it in the corresponding entry of the agents list. The method deactivate works the same way as the method suspend.

Reactivation of an agent

The method `activate` enables us to reactivate an agent that had been deactivated before. Such method is similar to the method `resume`, but in this case the agent is before loaded by the stream and then run from the beginning. Of course, the reference to the agent cancelled before by `deactivate` is adequately reassigned.

Killing an agent

If we do not want to continue the execution of an agent, it can be killed with the method `dispose`, that stops its execution (stopping the corresponding Thread) and cancels the corresponding entry from the list, so to delete all the references to such agent.

Migration of an agent

It is the most important aspect of the platform MAP, because it enables applications deriving from the class Agent to migrate through the network. The mechanisms needed for the migration of an agent (described before) are implemented within the primitive *go* of the Context. When an agent or a user recalls this method, the Context checks that the agent that must migrate is actually in the node and, once the Context finds it, serializes it, sends it to the destination, stops its Thread and deletes the corresponding entry in the vector of agents. Once the agent has reached its destination, the Daemon of the receiver node instances a new Instancer that reads the stream and recalls the appropriate methods of the Context, to instance the agent. Even in this case an agent can migrate only if it is not deactivated and if no other entity is acting on it.

Communication and Synchronization among agents

As we said before, the communication among agents takes place through message passing. Messages, as well as agents, travel in a serialized form. The Instancer, when receives a stream from the network, distinguishes whether it is an agent or a message through a boolean value inserted at the beginning of the stream, which takes a true value for the agent and false for the messages. Besides, in the case of messages, the reading of two other booleans from the stream enables us to make a distinction between synchronous and asynchronous messages, and between messages and requests of classes.

Exchanging messages among the agents is always managed by the Context In the case of asynchronous messages, the agent recalls the method `sendMessage` of the Context it belongs to, passing the message to be sent and the identifier of the receiver agent as parameters. At this point, the Context checks whether the receiver agent is local or remote; if it is a local agent, the Context inserts the message into the queue of messages of the agent. The receiver agent can extract each time (when it likes) the messages in the queue. Conversely, if the agent is remote, the Context sends the message to the remote platform that

contains that agent, whose address is specified in the message. Sent messages will be inserted into a queue from which the receiver agent can take them by using the method `getMessage`.

In the case of synchronous messages, the agent uses the method `sendSyncMessage` of the Context. The mechanism is the same as in the previous case. The difference is in the sender's behavior: in fact, in this case the agent waits for a reply message from the receiver agent. The receiver agent manages the messages sent to it through the method `receiveSyncMessage`.

5 Application to Network Management

In this section we will show how the MAP platform described can be successfully used in network management, overcoming some of the limits typical of a centralized approach. Current network management systems adopt a centralized paradigm according to which a protocol requires the management application to periodically access the data collected by a set of software modules located on network devices. There are, however, a large number of circumstances in which adoption of a distributed paradigm which can assign part of the control and management functions to the various network nodes is more appropriate [Gol93, GYM+95]. The basic idea is to reverse the logic according to which the data produced by the network devices is periodically transferred to the central network management station. If the management applications are encapsulated into the agents, it is possible *to port* them onto the network devices, thus performing a series of micromanagement operations locally and reducing the workload on the network management station and the overhead on the network as a whole. It is, in fact, reasonable to foresee the spread of network devices equipped with increasingly powerful local resources which will be able to reach a high degree of management sophistication, amply outperforming the reference models imposed by the platform-centered paradigm [GY95].

The basic components of a current network management system are:

- one or more management stations (*Network Management Station* or *NMS*);
- a (potentially large) number of nodes, each of which running a module called an *agent*, which monitors and collects the data for the node;
- a *management protocol*, used to transfer management information between agents and management stations.

The NMSs execute management applications which monitor and control network elements such as hosts, routers, terminal servers etc, by accessing their

management informaton. The latter is seen as a collection of *managed objects*, stored in *Management Information Bases (MIB)* [MR91]. Sets of correlated objects are defined in the MIB modules, which are specified using a subset of the standard OSI notation *Abstract Syntax Notation One (ASN.1)*, defined as *Structure of Management Information (SMI)*.

In the Internet environment, the *Simple Network Management Protocol (SNMP)* [ea90, Ros91] has become the standard protocol for network management. A network management protocol has to provide the primitives for the exchange of information between SNMP-agents and management stations. The set of SNMP primitives is relatively simple and offers three types of operations for the control of the various agents: the *set* and *get* operators to set or read the value of a variable and the *getnext* operator to examine the next variable in the MIB. SNMP-agents have a very simple structure and normally only communicate in response to requests for variables stored in the MIB. They cannot perform any management action on their local data.

The centralized paradigm adopted by the SNMP is appropriate in various network management applications, but the rapid increase in the size of networks has posed the question of the scalability of this or any other centralized model. At the same time, the calculation power of network nodes has also increased, making it possible to entrust them with significant distributed managament functions.

Centralization is generally appropriate for applications where the need for distributed control is low, frequent polling of MIB variables is not required and only a small amount of information is needed. A classical example is monitoring and viewing a few MIB variables. The status of a router interface, for instance, or the status of a link only entails querying and viewing a small number of MIB variables and centralized management is therefore suitable.

At the other extreme we have applications which require frequent polling of a large number of MIB variables, which have to perform calculations on a vast amount of information. An example would be calculation of a function indicating the level of functioning of the network, which requires very frequent detection of variations in a large number of MIB variables. In such cases monitoring and control should be performed as close as possible to the device in question.

Using MAP we can go beyond this distribution of tasks: the various management functions in our model do not have to reside statically on certain devices, but can migrate and dynamically execute on the particular node involved in the operation.

In order to use MAP for network management purposes, some additional

modules have been developed. As we are assuming that the platform is to be integrated with current management protocols, each node will need a standard SNMP agent to monitor the node. To use the data recorded by this agent, it will be necessary to use a set of classes implementing the SNMP communication protocol. These classes will thus be able to commmunicate with both local and remote SNMP agents. If these classes are present on each node, higher-level management applications can be constructed and implemented as agents. The system thus obtained is easy to extend: if the need arises to introduce a new management function specific to a certain subnetwork, it will be sufficient to develop it as an agent and it will be ready to be executed on any node belonging to MAP.

Application Example

Network management decisions have to be based on a vast amount of real-time data relating to the various devices: much of this data has to be suitably combined to give the system administrator all the available information regarding the functioning of the network in a simple, concise form.

In order to evaluate the effectiveness of the approach proposed as a valid support to the management decisions made by the system administrator, a simple application for the calculation of a function known in literature as the *health function* [GY95] has been implemented. In general the term *health function* refers to a linear aggregation of a number of management variables, each of which gives a particular measure regarding the device. In the SNMP environment these functions are typically a linear combination of MIB variables and the rate of exchange of these variables.

Such a function cannot be efficiently integrated in an SNMP-based network management model as the SNMP does not have the flexibility and decentralization required for its implementation. Although, in fact, it is possible to examine approaches based on *MIB metavariables*, with which these aims can be achieved, the functions to be calculated can often not be foreseen statically a priori. On the contrary, the aim is to be able to insert various kinds of functions according to the particular device to be monitored and its functioning conditions. Of the various functions that can be implemented, only some will be suitable for a particular device and will be used for limited periods during the functioning of the system; there is therefore no need to have all the functions present in each node in the network, nor do they need to be executed continuously.

Using MAP we are allowed to execute these functions directly on the node, thus meeting one of the main requirements for which they were introduced, i.e. compression of the amount of data to be processed directly at source. The

approach also allows the functions to be processed only when really necessary, thus avoiding the collection of a vast amount of MIB variables which will never be used completely.

The function used as an example is the following:

$$U(t_2, t_1) = \frac{((ifInOctets_2 + ifOutOctets_2) - (ifInOctets_1 + ifOutOctets_1)) * 8}{ifSpeed * (sysUptime_2 - sysUpTime_1)/100}$$

in which *IfInOctets* (*IfOutOctets*) represents the total number of bytes received (sent), *ifSpeed* is the nominal speed of the interface and *SysUpTime* is the time since the interface started working.

This function represents a measure of the average utilization of the network interface of a host, calculated during the interval $t_2 - t_1 = sysUpTime_2 - sysUpTime_1$. Taking measurements in which the time interval is minimal (i.e. $t_2 \to t_1$), the value of U tends towards an instantaneous value for the utilization of the interface.

By monitoring this function for the various network devices, the system administrator can gain a picture of the traffic generated on the various nodes. If, for example, the value for a specific node remains above a given threshold for long periods of time, this might indicate the presence of congestion in the network. By analyzing other functions, the administrator can understand the causes of the phenomenon and take the appropriate action.

We wish to emphasize that the function used only is to be considered as an example of application for our platform; it is clear that complete management applications will require a large, complete set of functions of this kind. The following considerations comparing the classical SNMP-based approach and our agent-based approach are of a general nature and do not depend on the particular function used.

In an approach based exclusively on the SNMP, to calculate the value taken by the management function it is necessary to perform continuous polling of the MIB variables involved. The traffic generated for a station will therefore be given by:

$$Traffic(byte/s) = \frac{n * l}{\Delta t}$$

where:

- n represents the number of MIB variables present in the function
- l represents the average length in bytes of the SNMP packets containing the values of the MIB variables required

- Δt is the variable sampling interval: its value depends on the type of function and the tradeoff between precision of the values calculated and the need to avoid saturating the network with excessively frequent polling

Various observations and measurements have shown that the values involved in the expression typically fall within the following ranges:

- $n = 3 \div 10$
- $l = 50 \div 400 byte$
- $\Delta t = 0.5 \div 5s$

Considering the worst case in which $n = 10, l = 400 bytes$ and $\Delta = 0.5$, we would get a network traffic of $Traffic = 8Kb/s$ which, as it refers to a single station and a single management function, is far from negligible. However, extreme values are only rarely reached: if we examine the most frequent values for these magnitudes ($n = 4, l = 200 bytes$ and $\Delta = 0.5$) we get a value of $Traffic = 1.6Kb/s$.

Obviously if this function has to be monitored for various stations the traffic generated by the management messages may be significantly high. If, for example, there are $N = 30$ nodes to be monitored, the average traffic due to the management processes would be $N * Traffic = 48Kb/s$.

Use of our approach eliminates all of this traffic; once the agent containing the application has been transferred on a remote node, the actual calculation will be performed locally with no need for communication with the central control station. Interaction between the agent on the remote node and the agent on the management station will only occur if certain specific situations arise: for example, in our particular case, if the function calculated exceeds a threshold for a long period of time a notification message will be sent back, which the manager can immediately examine or store in order to create a historical file of network events.

The only traffic generated using our approach is due to the transfer of the agent code from a node to another one on which it will be executed. The size of this code is quite limited: in the application implemented here, for example, the bytecode actually transferred is 4 Kbytes. Besides the small size of the code to be transferred, there is also the fact that the transfer is only made when necessary, and only for functions actually executed on remote nodes.

Use of MAP, however, does have a cost in terms of utilization of the computational resources of the network nodes, as a consequence of the remote execution of management functions.

The MAP server which always has to be active on the platform nodes execute in the background and as with any other daemon process, the only resource it consume in *sleep* periods is memory. It only becomes active when receives an agent or a message to deliver to some local executing agent. Significant measures for the processes involved in our platform are given in Table 1. They refer to a Sun SparcStation with the Solaris 2.4 operating system. The first column indicates CPU utilization, the second the percentage of system memory used by the process, the third the resident set size and the fourth the total size of the process.

	CPU%	MEM%	RSS (Kb)	SIZE (Kb)
MAP Server	0.0	5.3	≈ 1600	≈ 2800
MAP Agent containing a Typical Application	$10 \div 30$	$10 \div 15$	≈ 4500	≈ 5500

Table 1: Calculation Resources Used by Processes

When an application is being executed on a node, the CPU workload fluctuates with peaks reaching 20-30% utilization; the size in the memory grows according to the type of application, but is never high and comparable with common applications. In addition, subsequent applications on the nodes will execute with the same interpreter, that is, a new application will be executed as a new thread in the same process. This means that subsequent applications will not require execution of a new interpreter instance, thus reducing the amount of calculation resources consumed.

6 Future works and Conclusions

In this work we presented the design and the implementation of a platform for the development of mobile agents. We discussed the main designs issues concerning the mobility and the communication between agents, and we gave some implementation details. The language used for the development of the whole platform, as well as for the programming of agents, is Java. It was chosen for its features of portability on different architectures, multithreading, dynamic binding, possibility of dynamic loading of classes from different sources, etc.

We showed the different functionalities of the platform with the implementation of a fully distributed network management service. A comparison with the classical centralized approach is also provided. We described the applica-

tion created for managing the agents in a distributed system simply: thanks to such interface, we can easily create or suspend some agents, run them on remote nodes, check the state of agents in any remote server.

We are currently working to equip the MAP with adequate mechanisms of security and of access control. In particular, a SecurityManager will be placed between the Context and the agents requiring the services. Since each agent in MAP passes through the Context for accessing the system resources, the SecurityManager will be able to check the single actions of each agent. Each agent will have an owner and, according to this, will be associated with a list of authorizations about the resources that can be used. In general, an "anonymous" agent will be able to be associated with a set of standard authorizations limiting its actions, so not to allow the agent to do any action that might endanger the integrity of the system or the unauthorized access to remote information.

References

[BFD96] L.F. Bic, M. Fukuda, and M.B. Dillencourt. Distributed Computing Using Autonomous Objects. *IEEE Computer*, 29(8):55–61, August 1996.

[BHJL88] A. Black, N. Hutchinson, E. Jul, and H. Levy. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[CGH+95] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communications*, 2(5):34–49, October 1995.

[CHK95] D.M. Chess, C.G. Harrison, and A. Kershenbaum. Mobile Agents: Are they a Good Idea? Technical Report RC19887, IBM T.J. Watson Research Center, March 1995.

[CPV97] A. Carzaniga, G.P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proc. of the 19th Int. Conf. on Software Engineering (ICSE'97)*, May 1997.

[ea90] J. D. Case et al. A Simple Network Management Protocol (SNMP). *RFC 1157*, 1990.

[EW94] O. Etzioni and D. Weld. A Softbot-based interface to the Internet. *Communications of the ACM*, 37(7), July 1994.

[EW95] O. Etzioni and D.S. Weld. Intelligent agents on the Internet: Fact, Fiction, and Forecast. *IEEE Expert*, 10(4):44–49, August 1995.

[FMFM94] T. Finin, D. McKay, R. Fritzon, and R. McEntire. The KQML Information and Knowledge Exchange Protocol. In *Third Int. Conf. on Information and Knowledge Management (CIKM'94)*, November 1994.

[GK94] M. Genesereth and S. Ketchpel. Software Agents. *Communications of the ACM*, 37(7):48–53, July 1994.

[Gol93] G. Goldszmidt. On Distributed System Management. *Proceedings of the IFIP International Symposium on Integrated Network Management*, 1993.

[Gos95] J. Gosling. The Java Language Environment: a White Paper. Technical report, Sun Microsystems, May 1995.

[GV97] C. Ghezzi and G. Vigna. Mobile Code Paradigms and Technologies: A Case Study. In *Proceedings of the First Int. Workshop on Mobile Agents (MA97)*, Berlin, Germany, April 1997.

[GY95] G. Goldszmidt and Y. Yemini. Distributed Management by Delegation. *Proc. of the 15th International Conference on Distributed Computing Systems*, 1995.

[GYM+95] G. Goldszmidt, Y. Yemini, K. Meyer, M. Erlinger, J. Betser, and C. Sunshine. Decentralizing control and intelligence in network management. In *Proc. of the 4th International Symposium on Integrated Network Management*, May 1995.

[Ham96] M. A. Hamilton. Java and the Shift to Net-Centric Computing. *IEEE Computer*, 29(8):31–39, August 1996.

[JvRS95] D. Johansen, R. van Renesse, and F. Schneider. An Introduction to the TACOMA Distributed System. Technical Report 95-23, University of Troms and Cornell University, June 1995.

[KS97] J. Kiniry and D. Simmermann. A hands-on look at Java Mobile Agents. *IEEE Internet Computing*, 1(4):49–52, July 1997.

[LM96] W.W. Li and D.G. Messerschmitt. Java-To-Go: Itinerative Computing using Java. *http://ptolemy.eecs.berkeley.edu/dgm/javatools/java-to-go*, September 1996.

[LO97] D.B. Lange and M. Oshima. Programming Mobile Agents in Java with the Java Aglet API. Technical report, IBM Tokyo Research Division, 1997.

[Mag97] General Magic. Agent Technology: General Magic's Odyssey. *http://www.genmagic.com/html/agent_overview.html*, 1997.

[Mic97] Sun Microsystems. Java Object Serialization Specifications. *http://java.sun.com/products/jdk/rmi/serial*, 1997.

[MR91] K. McCloghrie and M. Rose. Management Information Base for Network Management of TCP/IP-based Internets: MIB-II. *RFC 1213*, 1991.

[MRK96] T. Magedanz, K. Rothermel, and S. Krause. Intelligent Agents: An Emerging Technology for Next Generation Telecommunications? In *Proceedings of INFOCOM'96*, San Francisco, CA, USA, March 1996.

[Obj97] ObjectSpace. Voyager Core Technology: Technical Overview. *http://www.objectspace.com/voyager*, 1997.

[PTV97] A. Puliafito, O. Tomarchio, and L. Vita. A Java-based Distributed Network Management Architecture. *Third International Conference on Computer Science and Informatics, CS&I'97*, March 1997.

[Ros91] Marshall T. Rose. Network Management is Simple: you just need the Right Framework. *Proceedings of the IFIP Second International Symposium on Integrated Network Management*, 1991.

[SBH96] M. Strasser, J. Baumann, and F. Hohl. MOLE - A Java Based Mobile Agent System. In *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, July 1996.

[SG90] J.W. Stamos and D.K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[SJRK97] K. Srinivas, V. Jagannathan, Y.V.R. Reddy, and R. Karinthi. Java and Beyond: Executable Content. *IEEE Computer*, 30(6):49–52, June 1997.

[TM95] R. Zahavi T.J. Mowbray. *The essential CORBA: Systems Integration Using Distributed Objects*. John Wiley & Sons, Inc., 1995.

[Way95] P. Wayner. *Agents Unleashed: A Public Domain Look at Agent Technology.* AP Professional, 1995.

[Whi94] J.E. White. Telescript Technology: The Foundation for the Electronic Marketplace. Technical report, General Magic, 1994.

[Whi95] J. White. Mobile Agents White Paper. Technical report, General Magic, October 1995.

[WPW+94] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet. Concordia: An Infrastructure for Collaborating Mobile Agents. In *Proceedings of the First Int. Workshop on Mobile Agents (MA97)*, April 1994.

[WWR97] A. Wollrath, J. Waldo, and R. Riggs. Java-Centric Distributed Computing. *IEEE Micro*, pages 44–53, June 1997.

[WWWK94] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report TR-94-29, Sun Microsystems Laboratories, November 1994.