A Performance Evaluation of the Mobile Agent Paradigm

L. Ismail
Institut National Polytechnique de Grenoble (INPG)

D. Hagimont
Institut National de Recherche en Informatique et en Automatique (INRIA)

Sirac Laboratory[1]
INRIA Rhône-Alpes, 655 avenue de l'Europe
38330 Montbonnot Saint-Martin, France

+33.4.76.61.52.66
Leila.Ismail@inrialpes.fr

+33.4.76.61.52.62
Daniel.Hagimont@inrialpes.fr

ABSTRACT

This paper presents a performance evaluation of the mobile agent paradigm in comparison to the client/server paradigm. This evaluation has been conducted on top of the Java environment, using respectively RMI, the Aglets mobile agents platform and a mobile agents prototype that we implemented. The measurements give the cost of the basic mechanisms involved in the implementation of a mobile agent platform, and a comparative evaluation of the two considered models (client/server and mobile agents) through two application scenarios. The results show that significant performance improvements can be obtained using mobile agents.

Keywords

Mobile agents, client/server model, distributed applications, performance, Java.

1. INTRODUCTION

Today, the most popular paradigm for structuring distributed applications is the client/server model. It is notably used as a base in the CORBA [11] and Java [6] environments through a remote object invocation mechanism.

Recently, a new paradigm has emerged for structuring distributed applications : mobile agents programming [2]. In this model, an agent is a process that can move with its own code and execution context from machine to machine in order to perform its task. Mobile agents are mainly intended to be used for applications distributed over large scale (slow) networks because they allow saving communication costs by moving computation to the host on which the target data resides [4].

While many researchers investigated the development of mobile agents systems, very few proposed an evaluation of the technology through actual measurement in a real large scale environment, along with an evaluation of the basic mechanisms involved in the implementation of such systems. A simple performance model was investigated in [1], and a performance model for applications using alternatively remote invocations and mobile agents was investigated in [13].

In this paper, we present a performance evaluation of the mobile agent model by comparison with the client/server model. This evaluation was performed in the Java environment using respectively RMI [15] (Java's remote object invocation mechanism), the Aglets mobile agents platform [9] [14] (a well-known mobile agents system) and a minimal mobile agents platform that we developed and for which we know the implementation details. The performance evaluation reveals the cost of the basic mechanisms involved in the implementation of a mobile agents platform on top of Java, and an evaluation of the two above models (client/server and mobile agents) through two applications scenarios.

The results of this evaluation show that:

- While the features provided by Java are well-suited to the implementation of mobile agents systems, these mechanisms are still very costly.
- Despite these costly mechanisms, significant performance improvements can be obtained thanks to mobile agents, especially when an agent which intensively communicates with a server moves to that server site.

The rest of the paper is structured as follows. In section 2, we present the principles of mobile agents programming, then we describe the mobile agents platform that we implemented and the measurements of the Java mechanisms involved in its implementation. Section 3 presents a comparative performance evaluation of the two considered models (client/server and mobile agents) through the two application scenarios that we selected. We conclude the paper in section 4.

[1] Sirac is a joint research laboratory between Institut National Polytechnique de Grenoble, INRIA, and Joseph Fourier University.

2. MOBILE AGENTS SYSTEMS

Many mobile agents systems have been developed in the last few years (Telescript [15], Aglets [9], AgentTcl [7], MOA [10]). Most of them have been implemented on top of Java, mainly for its wide spreading but also for its technical advantages: machine independence and strong typing for security.

In this section, we first recall the general features of mobile agents systems and then, after a review of the main Java mechanisms used in the implementation of a mobile agents platform, we describe the platform we implemented. We conclude this section with a quantitative evaluation of the main components of this prototype.

2.1 Features of a mobile agents platform

An *agent* is a mobile object that can move from machine to machine under its own control in order to achieve some tasks on these machines. In general, in order to move to a machine, an agent must explicitly invoke a *move(machine)* primitive.

An agent is composed of three parts:

- the agent code part which corresponds to a certain algorithm,
- the agent execution thread (with an execution stack),
- the agent data part which corresponds to the values of the agent's global variables.

The agent data can evolve dynamically as a consequence of visits to servers where information is collected. All these parts move with the agent whenever it moves.

When an agent moves from one machine to another, it should continue its execution on the destination node at the instruction which immediately follows the invocation of the *move* primitive. However, most of the mobile agents systems[2] provide weak migration, i.e. an agent moves with its code and data without the stack of the execution thread. This implies that an agent restarts execution from the beginning each time it moves to a machine. Consequently, the programmer has to include within the agent information which allows him to trace the agent execution state. Then, when the agent arrives at a destination machine, the agent algorithm has to verify this execution state so that the appropriate part of the agent code is executed on that machine.

In general, a mobile agents system provides primitives allowing mobile agents to communicate with each other and with the servers on the visited machines. These communication primitives take the form of message passing or procedure or method calls.

2.2 Involved Java mechanisms

Java is an object-oriented programming language which is used to generate programs that can be executed on a portable runtime environment (the Java Virtual Machine or JVM) that runs on almost every machine type.

Regarding mobility, the key mechanisms are the followings:

- Code mobility and portability. A key feature of Java is code mobility. This means that Java code can be moved between different machines. Such code mobility requires code portability. Code portability is provided by interpretation of *byte code*. A Java compiler can generate machine code which is common to (and independent of) any type of hardware and which is interpreted by the runtime of the language during execution. In order to manage code mobility, Java defines the notion of *class loader*. A class loader is an object which is in charge of the loading of classes from any data source, disk or network. A class loader is invoked by the runtime to translate a class name into the class Java reference. If the class is not already loaded in the class loader, the class is physically loaded (from disk or from a remote machine). More precisely, the class loader invoked to translate a class name included in class *C* is the class loader used to load *C*. Therefore, loading explicitly a class using a class loader implies that any class reachable from this class will be loaded by this class loader. Java class loaders allow classes to be loaded from any data source, and they allow the definition of multiple class name spaces (one per loader).
- Dynamic binding. Dynamic binding is a crucial point to code mobility. Dynamic binding means the ability to determine at run-time the code to be executed for a method invocation. Since Java allows classes to be dynamically loaded, a variable of a type (interface in Java) can include a reference that points to an object, whose class was loaded dynamically. Java postpones the binding of code (from this variable) until invocation time, thus allowing the execution of dynamically loaded classes.
- Serialisation. Java also provides an object serialisation feature [12] which allows instances to be exchanged between different runtimes. This feature provides a means for translating a graph of objects into a stream of bytes which can be sent as a message over the network or written into a file on disk. Each instance of a class which implements the *Serializable* interface can be serialised. Two inherited methods *writeObject* and *readObject* define, respectively, the default behaviour for serialising and de-serialising an object (i.e. write/read all the object fields to/from the stream of bytes). By default, all the objects that are referenced from a serialised object are serialised (they must implement the *Serializable* interface). In order to control the serialisation process, it is possible to override these methods to specify which fields of the object should be written and read. This allows controlling whether serialisation should propagate following an object reference field.
- De-serialisation. De-serialisation is the reverse process of serialisation. A graph of objects can be reconstructed from the byte stream which includes the serialised graph.

In the following section, we present how these mechanisms have been combined to implement our mobile agent platform.

2.3 A minimal mobile agents platform

For the purpose of our experiment, we have developed a minimal platform which provides the basic functions required to program mobile agents. We developed a new platform in order to identify the Java mechanisms involved in the implementation of a mobile agents system and to measure the cost of these mechanisms.

[2] Especially those built on top of Java which does not provide thread migration.

In our platform, an agent executes on a virtual machine that we call the *agent server*. An agent server provides an entry point which allows agents to be received, and an interface which allows agents to move to another agent server. An agent server generally provides incoming agents with one or several entry points to services (e.g. a database) which are represented by Java objects within the agent server. Mobile agents invoke a service by method invocations. When an agent arrives on a server, it can consult a symbolic name service on that server in order to obtain references to Java objects associated with the services available in the agent server. An agent server that wishes to make services available to incoming agents can use the name service in order to export references to Java objects associated with the services.

An agent is an instance of a user-defined Java class. This class must inherit from class *Agent*, and it must define a *main()* method which is the agent entry point. The platform implements weak migration. If the agent moves to another site, the *main* method is invoked on the agent object on the arrival site. The *Agent* class implements two methods: the *move(site, port)* method which allows an agent to move to a server identified by *site* (an IP address) and *port* (a TCP port on that machine), and the *back()* method which allows an agent to return to its origin server.

When an agent calls the *move* method, a message which includes the agent code and the agent data is constructed. The agent data are composed of a set of Java objects which are managed by the agent itself (a graph of objects). The serialisation mechanism of Java allows us to transform the objects graph into a byte array that we store in the message. Then the message is transferred via the network to the destination site using TCP/IP. The de-serialisation mechanism is used to recover the objects graph on the destination site. The class loading mechanism of Java (*ClassLoader*) allows network incoming classes to be dynamically loaded (i.e. transformed into executable classes). Since different agent programmers may use a same class name, a private class loader is associated with each agent in an agent server. This class loader is created when the agent arrives on the agent server. Notice that this class loader does not load classes on demand, but receives all the classes which constitute the agent code in a single message, the class loader being initialised with the agent's classes.

To summarise the design of our minimal platform, an agent migration to a server consists in the following steps:

a) Serialisation of the agent data and construction of a message which includes the agent's data and code.

b) Sending of the above message to the target agent server (we use a TCP connection).

c) Destruction of the agent in the origin agent server.

d) Reception of the message in the destination agent server.

e) Creation of a new thread for the execution of the agent .

f) Creation of a class loader associated with the incoming agent, this class loader being initialised with the agent code.

g) De-serialisation of the agent data.

h) Restart of the agent (with the *main* entry point)

2.4 Evaluation of the platform

2.4.1 Experimental environment

Since agents are mainly intended to be used for applications deployed over large-scale networks [8], our experiments were performed full-scale, i.e. on a set of workstations distributed over the Internet[3], but for comparison purpose, we made the same measurements over our local network. The two distributed environments we used have the following characteristics:

? Local network. It is composed of two Sun Ultra 1 workstations under Solaris, interconnected through a 10 Mbits Ethernet.

? Large scale network. It is composed of two comparable workstations interconnected through the Internet. The first machine is located in France (INRIA, Grenoble) and the second is located in United Kingdom (Queen Mary and Westfield College, London).

We used the JDK 1.1.5 for all the experiments described in this paper.

In order to provide more precise indications about the environments, we measured the basic capabilities of the networks and the JVM that we used (Table 1). The one-way trip latency measurements were performed using UDP and the bandwidth measurements using TCP. Notice that RMI relies on UDP and that mobile agents systems generally rely on TCP.

Latency on the local network	0.6 ms
Latency on the Internet	20 ms
Bandwidth on the local network	6.736 Mbits/s
Bandwidth on the Internet	1.032Mbits/s
Local Java method invocation	1 µs
Remote Java method invocation on the local network	3.1 ms
Remote Java method invocation in on the Internet	44 ms

Table 1. Basic costs in the experiment environment

This table shows the performance gap between the local network and the Internet. It also shows how Java basic mechanisms (local invocation, remote invocation) behave in the distributed environments that we used. Notice that remote invocation with RMI costs much more than a local invocation, which motivates the use of mobile agents in order to move computation closer to the target remote object.

2.4.2 Components costs within the platform

In order to evaluate the cost of each elementary mechanism within the platform, we decomposed the migration function into several components which correspond to the main steps presented in section 2.3. To perform these measurements, we developed a minimal agent which iterates a ping-pong migration between two

[3] The measurements were performed at night in order to limit the variations of the network performance.

machines. The agent size is 1475 bytes, including the code and the data parts of the agent.

The agent migration is composed of the following components:

? Agent serialisation. This phase corresponds to step a) in the description of the platform. A message is built, including the code and the data parts of the agent.

? Agent transfer. This phase corresponds to steps b), d) and e) in the description of the platform. The message is sent to the destination agent server. On the destination host, a thread is created and the message is delivered to this thread. Step c) is not included in this phase since the agent on the origin site is destroyed only after the message has been sent.

? Agent installation. This phase corresponds to steps f), g) and h) in the description of the platform. A new class loader is created for the incoming agent. The loader is initialised with the agent code and then the agent data are de-serialised.

The execution times for these phases are given in Table 2.

Agent Serialisation	3.2 ms
Agent transfer on the local network	8 ms
Agent transfer on the Internet	121 ms
Agent installation with the Java class loader	4.3 ms
Agent installation with a private class loader	23.8 ms

Table 2. Basic costs

These measurements show that the costs induced by Java are prohibitive compared to the cost of the agent transfer on the local network, but become acceptable when compared to the cost of the agent transfer on the Internet.

The installation phase includes the agent de-serialisation with a class loader private to the agent. Managing per-agent class loaders induces an overhead that we evaluated by measuring the cost of installing an agent under two conditions:

? Using Java's system class loader, which assumes that the agent classes are already installed on both machines. These classes are dynamically loaded at the first visit and kept in cache for subsequent visits.

? Using a newly created class loader for each visit of the agent (which iterates between the two machines). The classes are loaded from the code part of the incoming agent.

We observe that most of the cost of the agent installation comes from the agent classes loaded with a private class loader.

2.4.3 Overall cost of agent migration

To conclude this section, we measured the overall cost of agent migration in the two environments described above (local network and Internet), and we used a per-agent private class loader.

Minimal agent migration on the local network	35 ms
Minimal agent migration on the Internet	148 ms

Table 3. Overall cost of agent migration

First, notice that an agent migration costs much more than a remote object invocation. Therefore, moving an agent to a server must save a sufficient (large) number of remote communication between the client and the server in order to be beneficial. Section 3 illustrates this point with two application examples.

Our minimal platform only implements the basic functionality which allows programming mobile agents. In order to provide an indication of the cost of agent migration in a full-scale mobile agent platform, we performed the same measurements with the Aglets system [9] [14], a very well-known platform in this area (Table 4).

Minimal Aglet migration on the local network	240 ms
Minimal Aglet migration on the Internet	369 ms

Table 4. Migration cost in the Aglets platform

The difference with our platform is significant. It comes from the complexity of the protocol (Agent Transfer Protocol) used by Aglets, which requires several messages to migrate an agent.

Until now, we have described the implementation principles for building a mobile agents platform and we provided a performance evaluation of the main Java mechanisms involved in this implementation. In the next section, we study the interest of mobile agents programming, from a performance point of view, through a comparison with the client/server model.

3. COMPARISON WITH THE CLIENT/SERVER MODEL

In this section, we evaluate in which condition the mobile agents model can outperform the client/server model. After a description of the application domain that we consider, we present respectively, for each of the two selected application scenarios, its principle and the results of its evaluation.

3.1 Application domain

In the two application scenarios that we considered, mobile agents are used to adapt a service provided by a server according to specific requirements of the clients.

We suppose that an important motivation for servers is to provide a generic service which suits a large spectrum of requirements from all the potential client of the service. In order to provide a generic service, the server can export a parameterisable interface, but the complexity of this interface increases with the diversity of the clients needs. Another approach consists in decomposing the service into a set of lower level operations which can be invoked independently from each other. A client can combine invocations to these basic operations in order to implement the desired service.

The second approach is preferable since it provides a generic service without requiring to know in advance the clients needs. However, it implies more frequent interactions between the client and the server. In a distributed framework based on the client/server paradigm, remote interactions are costly and

genericity becomes therefore detrimental to the service's performance. A solution to this problem is to co-locate a part of the client application (the part which implements the desired service from the generic interface) and the generic service on the server site, in order to reduce remote interactions.

This co-location principle was the main motivation in the area of operating system extension (Spin [13], Exokernel [5]). Invoking a generic system service (in the kernel) requires many system calls which are very costly. An extensible system allows the execution of a part of the application in the address space of the operating system. Thus, invoking the service from this application part does not cross any address space boundary and is therefore much more efficient. We propose to apply this technique using mobile agents in order to install a part of the application on the server machine.

A client can develop a specialised service according to its specific needs, relying on the generic interface exported by the server. This specialised service can be sent to the server machine using a mobile agent, which implies that interactions between the specialised service and the generic service are local to the server machine. Figure 1 illustrates this scheme.

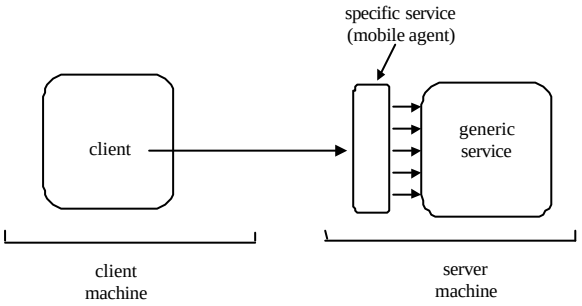


Figure 1. Server extension with a mobile agent

An important aspect of this scheme is the installation of the mobile agent on the server machine. There are two possibilities:

- Pre-installation. The specific service is installed by the client on the server before the service is invoked. After the installation of the specific service, the client can invoke the specific service through remote method invocations. The agent installation is performed once and remains valid for subsequent invocations.
- Dynamic migration. The client specific service is a mobile agent which migrates to the server site each time the service is invoked.

With a pre-installation of the agent, we only pay for agent migration when the specific service is installed. Subsequent invocations do not incur any overhead and benefit from the service specialisation. With dynamic migration, each service invocation is implemented as an agent migration. Notice that the installation of persistent agents on the server site may be denied by the server in order to prevent an accumulation of service extension on the server.

In the rest of the paper, we assume dynamic migration of agents because we aim at evaluating the interest of service specialisation depending on the cost of agent migration. In the pre-installation case and once the installation is performed, the benefit is obvious.

3.2 The *forward* application

3.2.1 The application

One of the important application domains for mobile agents is data mining where agents move among different machines in order to look for information on behalf of their clients.

Our example here is an application which looks for a list of hotels in a town. In our scenario, two databases must be consulted. The first database registers hotels and allows clients to obtain a list of the hotels in the town where the client is willing to stay. The second database is a telephone directory which allows clients to get the telephone number associated with a name[4]. We suppose that these two databases are managed on different machines and by different administrative entities (or different companies as for example a tourist office and a telephone company).

Each of these two servers autonomously provides an interface that corresponds to the proposed service. The interface of the first server allows clients, providing the name of a town, to obtain a table of the hotels in the town. The interface of the second server allows clients, providing a name, to obtain the phone number associated with that name.

Using the client-server model, in order to get the hotel list for a town including the hotels telephone numbers, the client first has to make one remote call to the first server (*A*) in order to get the hotel list. Once the hotel list is obtained, the client must make a remote call to the second server (*B*) for each hotel in the list in order to obtain the corresponding telephone number.

Consequently, the time to obtain the final response is equal to the time of the remote invocation from client *C* to *A* ($rpc_{C\text{-}A}(n)$, n being the number of hotels in the returned list), plus the time of the n remote invocations from *C* to *B* ($n*rpc_{C\text{-}B}(1)$, each invocation returning one record). This means that the overall cost is equal[5]:

$$total\ cost = rpc_{C\text{-}A}(n) + n*rpc_{C\text{-}B}(1)$$

Each server A and B provides a generic interface which is composed of simple operations (in this example, consulting the server). Then, clients can use A and B services through these generic interfaces according to their specific needs. However, client specific requirements are then translated into many interactions between the client and the servers, which affects applications performance. In this particular example, a join between the two databases is required by the client, which results in many RPCs.

A solution to the problem would be to have server *A* forward directly the hotel list to server *B* which would do the join and respond to the client. To implement this solution, we would need to specialise server *A* in order to forward the hotel list to server *B*. A static extension of server *A* (by the administrator of the server)

[4] Or any other complementary information: price fixed by a travel agency, quotation in a tourism book …

[5] The computation cost on the server can be considered negligible.

is not realistic since the two databases are managed separately. In addition, it is also not realistic to statically extend a server for a client specific need.

Consequently, we propose to use mobile agents to dynamically specialise server *A* in order to forward the hotel list (Figure 2).

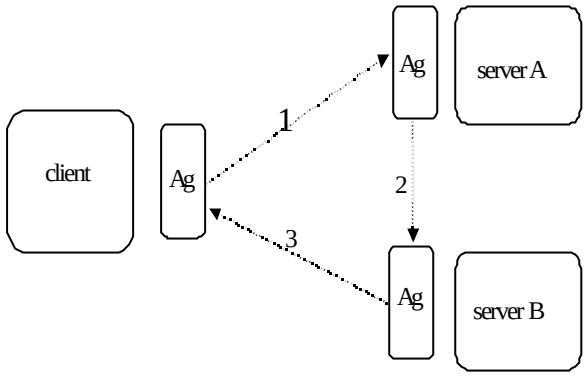


Figure 2. A mobile agent moves to forward the client request

The client creates an agent *Ag* which contains the client request to be executed. This agent moves (1) to server *A* for local interaction in order to obtain the hotel list. The hotel list is added to the agent data. The agent then moves (2) to server *B*, where it iterates on the list, respectively invoking *B* locally to obtain the telephone number for each hotel (this step corresponds to a join operation). The obtained telephone numbers are added to the data of the agent which moves (3) to its origin machine where it delivers the final result to the client[6].

We can notice through this example that mobile agents are a means for dynamically extending servers. Sending an agent to server *A* allows the implementation of request forwarding to server *B*. Now, let us look at the costs involved in the mobile agents implementation.

In this scheme, an agent with little data should be transferred from *C* to *A* ($Ag_{C-A}(0)$). Then, the agent moves to *B* with the hotel list ($Ag_{A-B}(n)$). Then, the agent returns to its origin machine with the list in which the telephone numbers are added ($Ag_{B-C}(2n)$)[7]. Here is the total cost:

$$total\ cost = Ag_{C-A}(0) + Ag_{A-B}(n) + Ag_{B-C}(2n)$$

Consequently, the mobile agents based implementation is beneficial if:

$$Ag_{C-A}(0) + Ag_{A-B}(n) + Ag_{B-C}(2n) < rpc_{C-A}(n) + n*rpc_{C-B}(1)$$

If we assume that functions *Ag()* and *rpc()* have a constant cost and if we use the measurements reported in section 2.4 for a minimal agent, we obtain that *n* should be more than 8 on the Internet. This is a rough estimate. In the next section, we present the results of the measurements on the forward application and give the value of *n* from which mobile agents are beneficial.

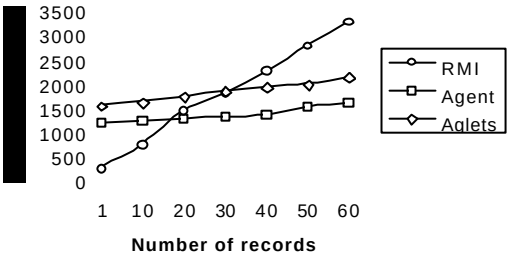
3.2.2 Measurements

In order to evaluate this application, we had to use a third machine compared to the environment described in section 2.4.1. This third machine is located at the University of Geneva (Switzerland). The following table (Table 5) summarises the capacities of our Internet environment.

Connection	Latency (ms) (one-way trip)	Bandwidth (Mbits/s)
France - United Kingdom	20	1.032
United Kingdom - Switzerland	16	2.24
Switzerland - France	26	0.984

Table 5. Capacities of the environment

As shown in the previous section, the response time depends on the number of records returned by the first server. Therefore, we varied the number of returned records and measured the cost of the overall query implemented with RMI, Aglets and Agent. In this experiment, the size of a record is 80 bytes. Figure 3 summarises the results.



As

Figure 3. Comparison between RMI and mobile agents for the forward application

We observe that, in our environment and for a very small number of records (less than 20 records), RMI is more efficient than a mobile agents based solution. This is explained by the fact that for a small number of records, the number of remote invocations that we save by using a mobile agent is not sufficient to amortise the cost of migrating an agent. However, with a sufficient number of records, the mobile agents based solution is much more efficient.

As previously mentioned, the difference between Aglets and Agent is due to the implementation of the transfer protocol, ATP being much more complex.

3.3 The *compress* application

The Web is mostly used in order to fetch documents. In this section, we consider an example in which a server exports an interface that allows a client to fetch a copy of a document. We

[6] The agent could directly send the result in a message to the origin site. We did not try to optimise this execution scheme.

[7] We assume that the size of the hotel list with the telephone numbers is 2n.

are especially interested in optimising transfer for large documents using data compression.

If the client wants to optimise the data transfer using a particular compression algorithm, it is necessary to extend the server in order to add the compression function. In addition, different clients may use different compression algorithms, which necessitates different extensions for different clients. As in the previous application example, this is not realistic.

The mobile agents paradigm offers an elegant response to this problem. Simply, the client can send an agent to the server site. This agent queries the server locally to obtain the document, compresses the obtained document, and returns to the client site with the compressed document. On the client site, the agent decompresses the document and delivers it to the client. This way, the client performs a dynamic extension of the server according to its specific needs[8].

The main motivation of the client for this server extension is the efficiency of the document transfer. In order to study the cost of this transfer, let us assume that the bandwidth between the two machines is *dr*. We suppose that the document compression factor is *fc* (if *S* is the initial size of the document, then the compressed document has a size of *fc***S*), and that the compression and the decompression times are respectively *tc* and *td*. We aim at identifying the network conditions under which transferring a compressed document is a gain for the application.

The transfer time of the compressed document is:

$tc + fc*S/dr + td$

Consequently, the compression is beneficial if:

$tc + fc*S/dr + td < S/dr$

This implies that compression is interesting if the network bandwidth between the machines such that:

$dr < S*(1 - fc) / (tc + td)$

This formula indicates the constraint on the bandwidth in order to potentially benefit from transferring a compressed document. In this formula, we did not take into account the cost of using a mobile agent. The results of our evaluation of this application are reported in the next section.

3.3.1 Measurements

For this experiment, we used documents (postscripts) whose sizes range from 100Kbytes to 2000Kbytes. We used the *gzip*[9] compression tool. For each document size, we measured the compression factor and the compression/decompression times. Substituting these values in the formula (previous section), we obtained that the compression is beneficial if :

- *dr* < *74* Kbytes/s for a document size of 100Kbytes,
- *dr* < *164* Kbytes/s for a document size of 500Kbytes.

We performed our measurements between the machine located in France and the machine located in United Kingdom. Taking into consideration the measured bandwidth (Table 1) between the two machines, the compression of a document can be beneficial in our Internet[10] experimental environment if its size is larger than 500Kbytes.

However, in our previous calculation, we did not take into consideration the cost of agents migration. Furthermore, the cost of migrating an agent is much higher than that of invoking a remote object with RMI. Therefore, it is difficult to predict whether the cost of agent migration can be amortised by the

communications saved thanks to compression.

Thus, we measured the cost of fetching the document with RMI without compression, and the cost of fetching the document with a compression agent (both with Aglets and Agent). We reproduced the experiment with different document sizes

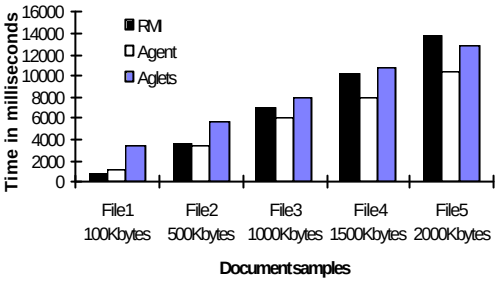


Figure 4. Comparison between RMI and Mobile Agents for the compress application

These results (Figure 4) show that for a small document (100Kbytes), RMI performs better than the two mobile agent platforms. However, for the 500Kbytes document, Agent performs better than RMI (so the compression benefit counterbalances the overhead of agent migration) while Aglets still performs worse than RMI. Finally, for larger documents, both mobile agents platforms outperform RMI, which means that when increasing documents size, the benefit of compression does balance the increasing overhead of agents migration compared to RMI.

The two application scenarios that we presented and evaluated show that specialising a remote service using a mobile agent is a means to obtain significant performance improvements compared to the client/server scheme, especially when the agent migration allows saving communications between the client and the servers.

4. CONCLUSION AND PERSPECTIVES

While many researchers investigated the development of mobile agents systems for programming distributed applications, very few proposed an evaluation of the technology through actual measurements in a real large scale environment. In this paper, we proposed such an evaluation of the mobile agents technology.

In order to measure more accurately the costs of the basic Java mechanisms involved in the implementation of a mobile agents

[8] This scheme can be used for data encryption.

[9] Accessible through a Java class.

[10] Notice that in our local network, we obtain a bandwidth of 842 Kb/s for which compression would slow down the request.

platform, we developed a minimal mobile agents platform which allows the development of testbed applications. We measured the costs of the main mechanisms and showed that, even if these Java mechanisms are very costly, their performance are still acceptable when considering applications distributed over the Internet.

In order to evaluate the interest of mobile agents programming compared to the client/server paradigm, we selected two applications which are characteristic of the application domain targeted by mobile agents. We implemented these applications in three ways, first using Java-RMI (which implements the client-server paradigm), second using our minimal platform and finally using the Aglets mobile agents system. The results of this experiment show that the use of mobile agents can lead to significant performance improvements, especially when the agent migration allows saving communications between a client and the invoked servers. For instance, in the forward application, the mobile agent allows remote invocations to be replaced by local invocations as the agent migrated to the invoked server sites. In the compress application, the agent allows the transferred document to be compressed and therefore to reduce the amount of data to be transferred.

The work presented in this paper is still under way and we are currently investigating in two directions.

The first perspective of continuation of this work is in the development and evaluation of larger applications using the mobile agent paradigm. We are currently using a mobile agents platform for the management of the Quality Of Service in a distributed multimedia application. This application consists in a browser which allows multimedia documents to be visualised. A multimedia document may include references to video or audio sources located on remote servers (as in an HTML page). The document may involve many multimedia sources (on servers) that the browser monitors, notably taking into account the bandwidth of the network. Obviously, it makes more sense to monitor a multimedia stream directly on the server machine rather than from the client machine, mainly for reactivity. We would like to use mobile agents in order to send the flow control part of the application close to the server.

The second perspective is to allow dynamic adaptation of the strategy used to interact with servers. We showed that specialising servers with mobile agents could be beneficial, depending on the speed of the network and the amount of communication between the client and the server. We would like to be able to dynamically reconfigure the application according to the environment in which the application is executed, and therefore to use the best strategy accordingly.

5. ACKNOWLEDGEMENTS

We would like to thank Jacques Mossière for his helpful comments on early drafts of this paper. Also, thanks to Jan Vitek and George Coulouris who accepted to provide us with login accounts on machines from their departments in Geneva and London respectively, in order to experiment with real distribution.

6. REFERENCES

[1] Carzaniga, A., Bian Pietro Picco, and Giovanni Vigna. Designing Distributed Applications with Mobile Code Paradigms. In Proceedings of the 19th International Conference on Software Engineering (ICSE'97), Boston (MA, USA), R. Taylor ed., May 1997, ISBN 0-89791-914-9, pp.22-32.

[2] Cockayne, William R., and Michael I. Zyda. Mobile Agents. Manning Publications Co., ISBN: 1-884777-36-8, 1997.

[3] Bershad, B.N., S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. 15th Symposium on Operating Systems Principles (SOSP), Copper Mountain Resort, Colorado, December 1995., volume II, Editor H.R. Arabnia, Las Vegas 1997, pages 1132-1140.

[4] Chess, D., C. Harrison, and A. Kershenbaum. Mobile Agents: are they a good idea ?. IBM Research Report, RC 19887, December 1994.

[5] Engler, D.R., M.F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. 15th Symposium on Operating Systems Principles (SOSP), Copper Mountain Resort, Colorado, December 1995.

[6] Gosling, J., and H. McGilton. The Java Language Environment: a White Paper, Sun Microsystems Inc., 1995. http://java.sun.com/whitePaper/java-whitepaper-1.html.

[7] Gray, R. A transportable agent system. In Proceedings of the CIKM Workshop on Intelligent Information Agents. Fourth International Conference on Information and Knowledge Management (CIKM95), Baltimore, Maryland, December 1995.

[8] Kotz, D., R. Gray, and D. Rus. Transportable Agents Support Worldwide Applications. In Proceedings of the Seventh ACM SIGOPS European Workshop, pages 41-48. ACM Press. September 1996.

[9] Lange, Danny, and Mitsuru Oshima. Programming and Deploying Java Mobile Agents with Aglets. Addison-Wesley Pub Co; ISBN: 0-201-32582-9, 1998.

[10] Milojicic, D. S., W. LaForge, and D. Chauhan. Mobile Objects and Agents (MOA). 4th USENIX Conference on Object-Oriented Technologies and Systems, Avril 1998.

[11] OMG. The Common Object Request Broker: Architecture and Specification, OMG Document Number 91.12.1, Revision 1.1, December 1991.

[12] Riggs, R., J. Waldo, A. Wollrath, and K. Bharat. Pickling State in the Java System. Computing Systems, 9(4), 1996.

[13] Straßer, M. and Markus Schwehm. A Performance Model for Mobile Agent Systems. Distributed

Processing Techniques and Applications DPPTA'97. Volume II, Editor H.R. Arabnia, Las Vegas 1997, pages 1132-1140.

[14] Venners, B. Under the Hood: The Architecture of aglets. March 1997. http://www.trl.ibm.co.jp/aglets/.

[15] White, J. E. Telescript Technology: The Foundation for the Electronic Market-place. General Magic Inc., Mountain View, CA, 1994.

[16] Wollrath, A., R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. Computing Systems, 9(4), pp. 291-312, 1996