Communicating Mobile Active Objects in Java

Françoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssière

OASIS Team, INRIA - CNRS - I3S
Univ. Nice Sophia-Antipolis
First.Last@inria.fr

Abstract. This paper investigates the design and implementation of mobile computations in Java. We discuss various issues encountered while building a Java library that allows *active objects* to migrate transparently from site to site, while still being able to communicate with each other. Several optimizations are introduced, and a set of benchmarks provides valuable figures about the cost of migration in Java: basic cost of migration with and without remote classloading, migration vs. standard remote method invocation in a typical information retrieval application. Our conclusion is that mobile computations are a viable alternative to remote method invocation for a large domain of Java applications that includes Web-based application.

1 Introduction

This paper is concerned with *mobile computations* [Car99]: the ability to start a computation on a site, suspend the execution of the computation at some point, migrate the computation to a remote site and resume its execution there. Sometimes called *mobile agents*, it is a far more complex problem than simply moving objects around as done in RMI or CORBA, or moving code alone as with Java applets.

A distinction is usually made between *strong migration* and *weak migration*. Strong migration means migrating a process by sending its memory image to a remote site: the current state of the stack, the value of the program counter, and of course all the objects reachable from the process. Moreover, the migration may occur preemptively; the process does not even need to know it has migrated. On the other hand, weak migration usually only involves the objects reachable from the process, and requires that the process has agreed to migrate (it is non preemptive). To the best of our knowledge, there exists no implementation of strong migration in Java that does not break the Java model or require user instrumentation of the code.

In this paper, we have taken the weak migration approach, and are trying to provide a flexible, extensible, yet efficient Java library for migration. As a result, we introduce a Java library for mobile computations allowing *active objects* to migrate transparently from site to site, while still being able to communicate with each other. It is built as an extension to the *ProActive PDC* library [CKV98].

Section 2 presents some related work in mobile agents technology and introduces the main features of the *ProActive PDC* library. Section 3 discusses and details the library for mobile computations, and presents some possible optimization techniques for mobile agents. Results from several benchmarks are presented in section 4, section 5 concludes.

2 Background on Mobility and Active Objects

2.1 Related Work

The best-known Java libraries for mobile agents are Aglets [Ven97] and Voyager [Obj99]. Both implement a form of weak migration, in the sense that an object must explicitly invoke a primitive in order to migrate, and this can only take place at points in the execution where a checkpointed state (either implicit or explicit) is reached. A *checkpoint* is a place in the code where the current state of the computation can be safely saved and restored.

For example, the `MoveTo` primitive of Voyager waits until all threads have completed. On the other hand, in Ajents [IC99], any object can be migrated while executing by interrupting its execution, moving the most recently checkpointed state of the object to a remote site and re-executing the method call using the checkpointed object state. Apart from the fact that whether checkpointing occurs lies in the hands of the user program, general and well-known issues related to checkpoint inconsistency due to rollback are kept unresolved.

Systems also differ in the way mobile objects interact: either by remote method call (Ajents, Voyager), or using a message-centric approach (Aglets). The default interaction mode is usually synchronous, even if some form of asynchronous communication is sometimes provided (in Aglets or Ajents for instance). Interacting synchronously simplifies the overall management of migration: while a remote method or message is handled, nothing else can happen to the two partners (and especially no migration), but this of course incurs a performance penalty.

Remote interaction is achieved transparently by a proxy which hides the effective location of the destination object to the caller. The proxy also often acts as a forwarder for locating the mobile object (see section 3.2) and is a convenient place for performing security-related actions.

Protocols for managing the migration of an object introduce several events, typically events such as *departure* or *arrival*, which can be customized by the user, like for example with Aglets. It is also possible to provide the mobile agent with an itinerary, i.e. a list of hosts to visit.

2.2 Asynchronous Active Objects

The results presented in this paper capitalize on research done over the last few years around the *ProActive PDC* library [CKV98]. *ProActive PDC* is a Java library for concurrent and distributed computing whose main features are transparent remote active objects, asynchronous two-way communications with transparent futures and high-level synchronization mechanisms. *ProActive PDC* is built on top of standard Java APIs (Java RMI [Sun98b], the Reflection API [Sun98a]). It does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, preprocessor or modified virtual machine.

A distributed or concurrent application built using *ProActive PDC* is composed of a number of medium-grained entities called *active objects* which can informally be thought of as “active components”. Each active object has its own thread of control and has the ability to decide in which order to serve incoming method calls. There are no shared passive objects (normal objects) between active objects, which is a very desirable property for implementing migration.

Method calls between active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism called *wait-by-necessity.*

As both active objects and future objects are type-compatible with the equivalent 'normal' objects, the programmer does not need to modify any code when reusing old code with ProActive. The only code that needs to be changed is the code that instanciates the objects we now want to be active. Here is a sample of code for creating an active object of type `A`:

```
class pA extends A implements Active{}
Object[] params={"foo", 7};
A a = (A) ProActive.newActive ("pA", params, node);
```

A full description of the library is outside the scope of this paper and can be found in [CKV98].

3 Communicating and Asynchronous Mobile Objects

Enhancing active objects with mobility is a nice feature to have but it is not enough: mobile active objects also need to be able to communicate with each other, regardless of where they are.

In this section we present the mechanism we have built into *ProActive PDC.*

3.1 Programmer Interface

The principle is to have a very simple and efficient (optimized) primitive to perform migration, and then to build various abstractions on top of it.

Primitives. Any active object has the ability to migrate and if it references some passive objects, they will also migrate to the new location. That is, we not only migrate an active object but also its complete subsystem. Since we rely on the serialization to send the object on the network, the active object must implement the Serializable interface. Many different migration primitives are available, all with the same general form :

```
    public static void migrateTo(...) throws ...
```

Notice that we are only able to perform weak migration since we do not have access to the execution stack of the JVM. Hence, a migration call must be encapsulated into a method of the mobile object so that in the end the call comes from the object itself. This call never returns because, strictly speaking, the object is not present anymore and so the execution must stop.
There are two different ways to move an active object in *ProActive PDC*. The first one consists in migrating to a remote host. In order to do so, the remote host must have a running Java object called a *Node*, a kind of daemon in charge of receiving, restarting active objects and keeping trace of locally accessible active objects.
The other way to migrate is to join another active object on a remote host, even if the address of this host is unknown. Indeed, only a reference to the remote object is needed. However it can not be guaranteed that after the migration the two objects will be on the same host, the referenced object having possibly migrated. An example of a very simple mobile object can be found in example 3.1. For the sake of clarity, this code has been simplified and exception handling is not shown.

Exemple 3.1 SimpleAgent

```
public class SimpleAgent implements Active, Serializable {

  public void moveToHost(String t)
    {
      ProActive.migrateTo(t);
    }

  public void joinFriend(Object friend)
    {
      ProActive.migrateTo(friend);
    }

  public ReturnType foo(CallType p)
    {
      ...
    }
 }
```

Higher Level Abstractions. With this primitive for migrating active objects from host to host, we can build an API on top of it in order to implement autonomous active objects, also known as mobile agents in the distributed computing literature.

Itinerary. We want an autonomous object to be able to successively visit all the sites on a list and perform a possibly different action on each of them (see [KZ97]). Such a list does not need to be known at compile-time and can be dynamically modified at runtime in order to react to changing environmental conditions.

Any active mobile object in *ProActive PDC* can have an itinerary, which is a list of <*destination, action*> pairs, where *destination* is the host to migrate to and *action* is the name of the method to execute on arrival. Therefore, the method to execute on arrival at a host differs from host to host and can be modified dynamically.

Automatic Execution. In some Java libraries for mobile agents, such as Aglets [Ven97], only one predefined hard-wired method to be called on arrival is allowed. This is an annoying limitation, since no decision on which method to call on arrival can be made at runtime.

The approach we have chosen makes use of reflection techniques in order to bring more flexibility in choosing the method to execute on arrival. Setting the method to execute on arrival is done through this call:

ProActive.onArrival(String)

There are two limitations on the signature of the method to be executed on arrival. First, the method cannot return any value since it is called by an internal mechanism and there is nobody to return the result to. Second, the method cannot have any parameter. Remember that this method will be executed when the mobile object arrives on a new site, so there might be a delay between the moment the method is set and the moment it is called. The value of the parameters cannot be known for sure, and so, proving properties on the system can be difficult. Therefore, the signature of the method to execute on arrival must be:

void myMethodToExecuteOnArrival()

This is not in any case a limitation. The method can call other methods and access attributes of the object as would do any other method.
There is also the event corresponding to the start of a migration:

ProActive.onDeparture(String)

The method specified with onDeparture() could be used to do some clean-up once the mobile object has left the local host.

3.2 Rationale and Discussion on Implementation Techniques

Implementing mobility features into *ProActive PDC* raised a number of interesting issues. One of them is how do we make sure that a mobile active object always remains reachable, even if it never stops jumping from one host to the next ?

In this section we survey two of the most popular solutions to the *location problem*. The first solution is based on a *location server*, chain of *forwarders*. Other solutions exist as the one described in [Bru99] that uses a distributed two-phase transaction and a sophisticated reference-tracking mechanism.

Location Server. The location server responsability is track the location of each mobile active object: Every time an active object migrates, it sends its new location to the location server it belongs to. As a result of the active object leaving a host, all the references pointing to its previous location become invalid. There exists different strategies for updating those dangling references with the new location, one of them being lazy: when an object tries to send a message to a mobile active object using a reference that is no longer valid, the call fails and a mechanism transparently queries the location server for the new location of the active object, updates the reference accordingly and re-issues the call (see figures 1 and 2).

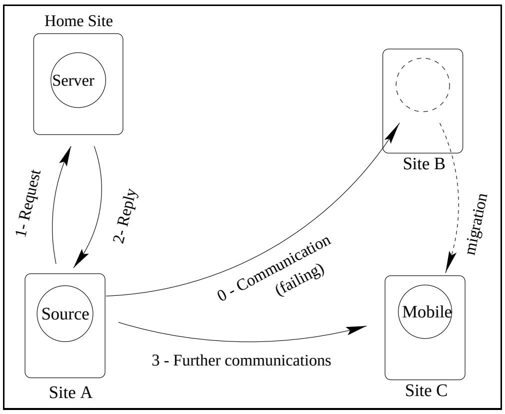

Fig. 1. Localisation - the caller side

Being a centralized solution, it is very sensitive to network or hardware failures. Standard techniques for fault-tolerance in distributed systems could be put to use here, such as using a hierarchy of possibly replicated location servers instead of a single server. Nevertheless, this solution is costly, difficult to administer and would certainly not scale well.

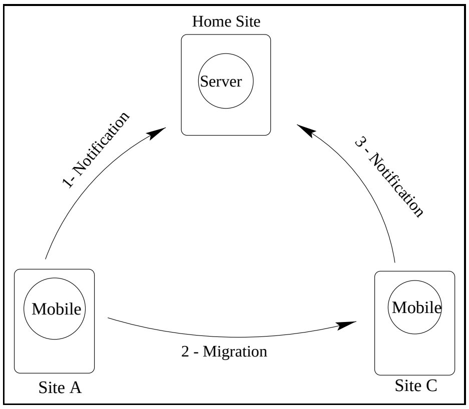

Fig. 2. Localisation - the mobile object side

Forwarders.

An alternative solution is to use *forwarders* [Fow85]: knowing the actual location of a mobile object is not needed in order to communicate with it; rather, what really matters is to make sure that the mobile object will receive the message we send to it.
To do so, a chain of references is built, each element of the chain being a *forwarder* object left by the mobile object when it leaves a host and that points to the next location of the mobile object [Obj99]. When a message is sent, it follows the chain until it reaches the actual mobile object. This appears to be the solution of choice for several systems [KZ97].
In *ProActive PDC*, for efficiency purpose, it is the active object that is turned into a forwarder object at the moment it leaves a host. This mechanism is fully transparent to the caller because the forwarder has exactly the same type as the mobile object. Moreover, the same mechanism is used for sending asynchronous replies to active objects, which means that an active object can migrate with some of the future objects in its subsystem still in the *awaited* state.

Forwarders can be considered as a distributed solution to the location problem, as opposed to the previous centralized solution. Moreover, contrary to the location server, in the absence of network or host failure, a message will finally reach any mobile object even if it never stops migrating. However, this solution also suffers from a number of drawbacks.
First, some elements of the chain may become temporarily or permanently unreachable because of a network partition isolating some elements from the rest of the chain or just because a single machine in the chain goes down; it would destroy all the forwarder located on it[1]. The chain is then broken, and it be-

[1] Except if the forwarders are persistent objects.

comes impossible to communicate with the mobile object at the end of the chain, although it is still well and alive.

Shortcutting the Chain of Forwarders. We present here a technique to keep the length of a forwarding chain to the minimum to limit the consequence of a break, as done in [Obj99]. It is based on the simple remark that the shorter the chain, the better. We take advantage of two-ways communication, i.e invocation of methods that return a result. When the object that executes the remote call receives a request, it *immediately* sends its new location to the caller if and only if it does not know it yet. Messages that are forwarded are marked so that an object receiving them knows if the sender has its current location or an older one. Thus we can avoid sending unnecessary update messages.

4 Benchmarks

Cost of one Migration. The first benchmark has been conducted to evaluate the cost of the migration between different computers.

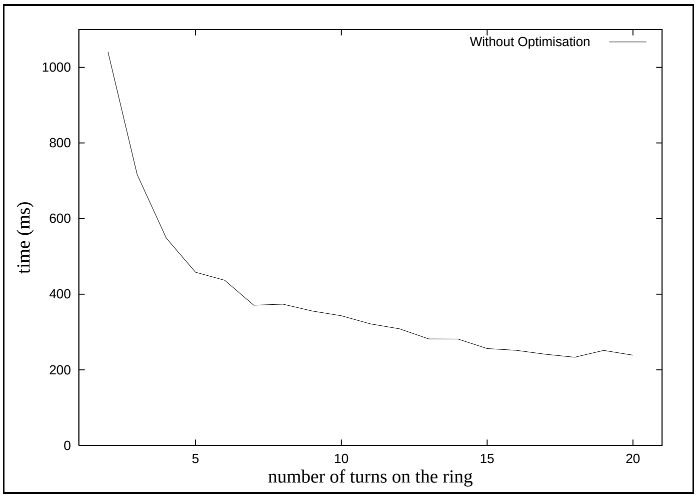

Fig. 3. Cost of one migration

Our test network was made of 2 Sparc-10, one Pentium-Pro, and a Bi-PentiumII. The program consisted in a mobile object moving from one host to another, always in the same order and always ending on the one it first started. It is equivalent to having the mobile object make one or more turns on a virtual ring of computers. The object is initially out of the ring, and so has to migrate first before actually starting its journey. We have measured the time taken to go through the whole itinerary, including the initial migration. The start-up time (first creation of all objects) has not been taken into account. Figure 3, shows the average time of one migration against the number of turns on the ring.

As time goes, the average cost of a migration decreases. This can easily be explained by the fact that on the first round on the ring, all the JVMs must load the class corresponding to the mobile object: when benchmarking mobile objects system, one should not forget that there is always a start-up time if the host is visited for the first time, even if all the class files are available locally or through NFS.
We can infer that the average migration time should be around 200ms on a LAN for some given network conditions. Keep in mind that these results are heavily dependent on the computating power.

Information Retrieval. The second benchmark has been designed to test a common application of mobile code: information retrieval. A text file representing the directory listing of a FTP server [2] is available on a remote host and we want to perform a search on this file to find all lines containing a certain keyword (i.e we are looking for a file). Without code mobility, we must first download the file (using a remote call on the active object that manages the file) and then perform the search; a mobile object would simply migrate to the remote host, perform the search and only bring the resulting lines.

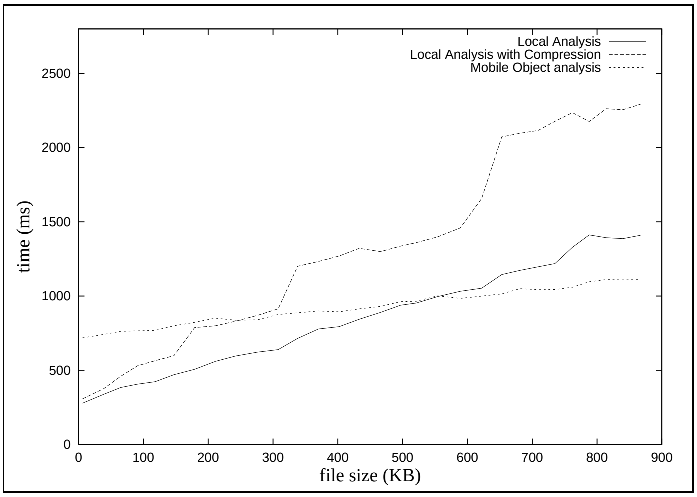

Fig. 4. Execution time of a file search

The purpose of this benchmark is to find the threshold above which migration becomes a better solution than local analysis of the downloaded file. Two

[2] `ftp.lip6.fr`

versions have been designed, both using the same methods for reading the file from disk and analysing it. In order to experiment with potential optimisation in the remote method call version, we have implemented "on the fly" compression of the downloaded file with the Java compression facilities (ZipInputStream). The test has been conducted on a 100 Mbits LAN with a Bi-PentiumII and a Bi-PentiumIII.

Figure 4 shows the result of 3 experiments, one with a mobile object, and two using local analysis. In the latter case, the file has been downloaded with and without compression to measure the possible gain.
Contrary to what was expected, compression of the file didn't decrease the time taken to perform the search. Even worse, the difference increases as the file size grows. Because the computers are on a high bandwidth LAN, the cost of compressing data on the fly is greater than the time saved on transmission. Also, we can notice that at file sizes 180, 350 and 650 KB, the time suddenly increases. We believe this is due to the algorithms used to compress the data and to related memory managment operations.
The mobile object program shows a remarquable scability since its execution time increases only slightly with the size of the file. Most of the time is spent performing migration (one to the remote host, and one back to the home host) as we can see from the time taken with a small file. Even on a high bandwidth and low latency LAN, mobile objects can be useful in many ways. First they reduce the load on the network and the hosts, and second their execution time can be much lower than conventional search, even for common sized files.

5 Conclusion

In this paper we have shown that mobile computation in Java can be efficiently implemented without breaking the standard Java execution environment in any way. The cost of one migration in our system is around 200 ms which is of the same order of magnitude as a remote call across the Web today. This means that using mobile computations instead of remote method invocations for client-server computing can quickly deliver benefits in application domains such as e-commerce or information retrieval.

Moreover, our implementation enables mobile active objects to communicate using two-way asynchronous message-passing, which further improves performances over other libraries for mobile computations.

Our future work will be aimed at studying mechanisms for fault-tolerance and security in mobile computation systems. The *ProActive PDC* library, its extension for mobile computations discussed here and the code for all the examples are freely available for download at `http://www.inria.fr/oasis/proactive`.

References

Bru99. E Bruneton. Indirection-Free Referencing for Mobile Components. In *Proc. of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Madeira Island, Portugal, April 1999.

Car99. Luca Cardelli. Abstractions for mobile computation. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603:51–94, 1999.

CKV98. D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.

Eck98. Bruce Eckel. *Thinking in Java.* Prentice Hall, 1998.

Fow85. Robert Joseph Fowler. *Decentralized Object Finding Using Forwarding Addresses.* PhD thesis, University of Washington, 1985.

HP99. B. Haumacher and M. Philippsen. More efficient object serialization. In *Parallel and Distributed Processing, International Workshop on Java for Parallel and Distributed Computing*, pages 718–732, San Juan, Puerto Rico, April 1999. Springer-Verlag. LNCS 1586.

IC99. M. Izatt and P. Chan. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *Proc. of the 1999 Java Grande Conference* . ACM, 1999.

KZ97. K. Kiniry and D. Zimmerman. A hands-on look at java mobile agents. *IEEE Internet Computing*, 1(4):21–30, July/August 1997.

Obj99. ObjectSpace, Inc. ObjectSpace Voyager `http://www.objectspace.com/developers/voyager/index.html`, 1999.

Sun98a. Sun Microsystems. Java core reflection, 1998. `http://java.sun.com/products/jdk/1.2/docs/guide/reflection/index.html`.

Sun98b. Sun Microsystems. Java remote method invocation specification, October 1998. `ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf`.

Ven97. Bill Venners. Under the hood: The architecture of aglets. *JavaWorld: IDG's magazine for the Java community*, 2(4), April 1997.