Mobile objects in Java

Luc Moreau[a] and Daniel Ribbens[b]
[a]*Electronics and Computer Science, University of Southampton, SO17 1BJ Southampton, UK*
Tel.: + 44 23 8059 4487; Fax: + 44 23 8059 2865; E-mail: L.Moreau@ecs.soton.ac.uk
[b]*Service d'Informatique, University of Liège, 4000 Liège, Belgium*
Tel.: +32 4366 2640; Fax: +32 4366 2984; E-mail: ribbens@montefiore.ulg.ac.be

Abstract: *Mobile Objects in Java* provides support for object mobility in Java. Similarly to the RMI technique, a notion of client-side stub, called startpoint, is used to communicate transparently with a server-side stub, called endpoint. Objects and associated endpoints are allowed to migrate. Our approach takes care of routing method calls using an algorithm that we studied in [22]. The purpose of this paper is to present and evaluate the implementation of this algorithm in Java. In particular, two different strategies for routing method invocations are investigated, namely *call forwarding* and *referrals*. The result of our experimentation shows that the latter can be more efficient by up to 19%.

1. Introduction

Over the last few years, mobile agents have emerged as a powerful paradigm to structure complex distributed applications. Intuitively, a mobile agent can be seen as a *running* software that may decide to suspend its execution on a host and transfer its state to another host, where it can resume its activity. Cardelli [5] argues that mobile agents are the right abstraction to develop applications distributed across "network barriers", e.g. in the presence of firewalls or when connectivity is intermittent. In Telescript [17], software migration is presented as an alternative to communications over a wide-area network, in which clients move to servers to perform computations. Lange [16] sees mobile agents as an evolution of the client-server paradigm, and enumerates several reasons for using software mobility.

It is a challenge to design and implement mobile agent applications because numerous problems such as security, resource discovery and communications, need to be addressed. Therefore, we introduce *Mobile Objects in Java*, a middleware that helps implement mobile agent systems by providing a concept of mobile object. Its specific contribution is a communication mechanism consisting of the invocation of methods on objects that may be mobile.

Our motivation has been driven by developments in distributed computing over the last couple of decades. Successive paradigms such as remote procedure calls (RPC) [3], method invocation in Network Objects [4] and remote method invocation (RMI) in Java [12], amongst others, abstract away from the reality of distribution. They successively provided programmers with new and more sophisticated abstractions. RPC provides *homogeneity*, by its marshalling and unmarshalling of data structures using data representation suitable for heterogeneous platforms. Network Objects offers *memory uniformity* because remote method invocations are syntactically identical to local ones and garbage collection takes care of local and distributed objects. Java RMI provides *code propagation* because the programmer no longer needs to replicate code to remote machines, but instead Java RMI is able to load code dynamically.

The next logical step is to hide the *location and movement of objects*. A similar approach has also been adopted by the network community, which devised the next generation of the IP protocol (IPv6) with support for mobile addresses [14].

There exists an incremental approach to introduce mobility into an infrastructure that is unaware of mobility [7,16]. It consists of associating each mobile entity with a stationary *home agent*, which acts as an intermediary for all communications. While this approach preserves compatibility with an existing infrastructure, introducing an indirection to a home agent for every communication puts a burden on the infrastructure; this may hamper the scalability of the approach, in particular, in massively distributed systems, such as the amorphous computer [27] or the ubiquitous/pervasive computing environment [1]. Free from any compatibility constraint, we adopted an algorithm to route messages to mobile agents that does not require any static location: the theoretical definition of this algorithm based on forwarding pointers and the proof of its correctness have been investigated in a previous publication [22].

The purpose of this paper is to present *Mobile Objects in Java*, an implementation of the algorithm, which offers transparent method invocation and distributed garbage collection for mobile objects. By transparent, we mean that mobile and non-mobile objects present a same interface, which is independent of the object location and its migratory status. Distributed garbage collection ensures that an object, whether mobile or not, can be reclaimed once it is no longer referenced. While implementing our algorithm, it became clear that two strategies could be adopted, which we named *call forwarding* and *referrals*; we present these strategies and evaluate their performance through a benchmark.

This paper is organised as follows. First, we provide more motivation for mobile agents by presenting two promising application domains in Section 2. Then, in Section 3, we summarise the algorithm we have investigated in [22]. In Section 4, we describe its implementation in Java, providing a transparent interface to mobile objects. We then discuss two different methods for routing method invocations, namely forwarding and referrals, in Section 5. In order to compare these techniques, we devise a synthetic benchmark, and analyse results in Section 6. Finally, we compare our approach with related work in Section 7.

2. Motivation

In this Section, we provide further motivation for mobile agents. We describe two promising applications where mobile agents act semi-autonomously on behalf of users. The reasons for doing so, however, differ substantially in the two applications.

Digital library

Yan and Rana [30] present a high-level service for a digital library of radar images of the Earth. The library is composed of a set of confidential images and associated annotations with attached ownership. They extend a Web-based client-server architecture with mobile agents that perform tasks on behalf of users and that are able to migrate to a predefined itinerary of hosts.

After being dispatched, agents migrate securely, with data, code and state to an itinerary of servers that may have relevant data and services. Agents become independent of the user who created them: they can survive intermittent or unreliable network connections. Mobile agents are beneficial for several reasons.

(i) They avoid the delivery of large volume of scientific data required for data mining of images;
(ii) They help maintain confidentiality and ownership of data, by being run through security checks, ensuring that they have the rights to access the data;
(iii) They are allowed specific queries on the library according to the "security level" they were granted.

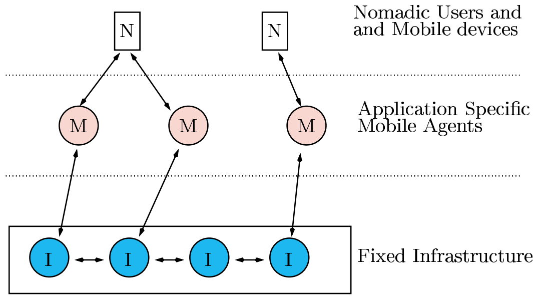

N
N
Nomadic Users and and Mobile devices
M
M
M
Application Specific Mobile Agents
I
I
I
I
Fixed Infrastructure

Fig. 1. Architecture.

Mobile users

The context of the Magnitude project [24] is the "ubiquitous computing environment" [27] where embedded devices and artifacts abound in buildings and homes, and have the ability to sense and interact with devices carried by people in their vicinity. Applications running on mobile devices interact with the infrastructure, and find and exploit services to fulfill the user's needs.

However, communications between mobile devices and the infrastructure have some limitations, in the form of intermittent connectivity and low bandwidth. Furthermore, processing power and memory capacity of compact mobile devices remain relatively small. As a result, such an environment would prevent the large scale deployment of advanced services that are communication and computation intensive.

We adopt mobile agents as proxies for mobiles users. As illustrated by Fig. 1, we utilise mobile agents, as semi-autonomous entities, which can migrate from mobile devices to infrastructure locations to take advantage of the resources their specific tasks require; mobile agents perform their tasks on the infrastructure, possibly involving further migration, and then return results back to mobile users.

Summary

Both scenarios use the idea of mobile agent, as a semi-autonomous proxy for a user. If granted the right to do so, mobile agents may migrate to new locations, where they can take advantage of local resources.

3. Message routing algorithm

In this section, we summarise the message routing algorithm for mobile agents that we formalised in [22]. We consider a set of mobile objects and a set of sites (in our case JVMs) taking part into a computation. Each mobile object is associated with a timestamp, which is a counter incremented every time the object changes location. Each site keeps a record of the location where every mobile object known to the site is thought to be, and of the timestamp the object had at the time. Therefore, in a system composed of several sites, sites may have different information about a same mobile object (depending on how fast location information is propagated between sites).

The algorithm proceeds as follows. When a mobile object decides to migrate from a site A to another site B, it informs A of its intention of migrating; a transportation service is used to transport the object to B. When the mobile object arrives at B, its safe arrival is acknowledged by informing its previous site A of its new location and of its new timestamp; site A can then update its local table with the mobile object's new position and timestamp.

Mobile objects delegate to sites the task of sending messages to other objects. When a site receives a request for sending a message to a mobile object, it searches its table in order to find the object location. If the object is local, the message is passed onto the object. If the object is not local, but known to be at a remote location, the message is forwarded to the remote location.

As migration is not atomic, a mobile object may have left a site, but the acknowledgement of its safe arrival may not have been received by the site yet. In such a case, the site temporarily has to enqueue messages aimed at the object; as soon as the acknowledgement arrives, delayed messages may be forwarded.

Timestamps are used to guarantee that sites always update their knowledge about mobile objects with more recent information than the one they currently have. If a site receives information with a timestamp that is smaller than the timestamp in its table, the received information is discarded. Such a timestamp mechanism is mandatory to avoid cyclic routing of messages [22].

In the algorithm described so far, a mobile object leaves a trail of forwarding pointers during its migration. In order to reduce the length of the chain of forwarding pointers, routing information and associated timestamp may be propagated by any site to any site; timestamps are again used to guarantee that the most recent information is stored in routing tables. In the rest of the paper, we discuss an implementation of this abstract algorithm.

4. Implementation in Java

In Java RMI [12], an object whose methods can be invoked from another JVM is implemented by a *remote object*. Such a remote object is described by one or more remote interfaces. Remote method invocation is the action of invoking a method of a remote interface on a remote object. In practice, a *stub* acts as a client's local representative or proxy for a remote object. The stub of a remote object implements the same interface as the remote object: when a method is called on the stub, arguments are serialised and communicated to the remote object,[1] where the method can be called; its result is transmitted back to the stub and becomes the method call result. A very desirable feature of this approach is that local and remote method invocation share an identical syntax.

Now, we present an approach in which remote objects are allowed to be mobile, but clients still use the same stub-based method invocation mechanism, making them unaware of the location and movement of the mobile object.

4.1. Startpoints and endpoints

Figure 2 displays the different entities of our implementation. The right-hand side of the picture repre-

[1]Before Java 1.2, there was a notion of skeleton, which was a server-side representative of the object, responsible for deserialising the arguments.

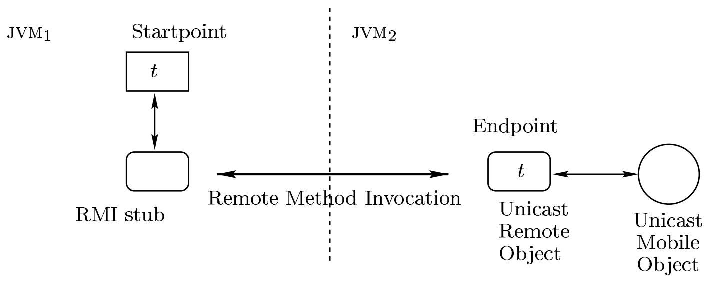


Fig. 2. Startpoint and endpoint.

sents the "server-side" on JVM_2, composed of a mobile object; the left-hand side is concerned with the "client-side" on JVM_1.

We adopt Nexus terminology [8], and we respectively name *startpoint* and *endpoint* the client-side and server-side representatives of a remote object. A mobile object is specified by an interface, which must also be implemented by its startpoints. Startpoints contain an RMI stub representing the current location of a mobile object, and permit direct communication with the endpoint; the endpoint passes messages to the mobile object. Additionally, the startpoint contains the mobile object's timestamp t. (The endpoint also has the same timestamp t.)

Figure 3 displays the new configuration after the mobile object has migrated to JVM_3. There exists a new endpoint acting as a server-side representative at the new location. Its timestamp is $t + 1$ following its increase after migration. The endpoint is referred to by a startpoint with timestamp $t+1$, which is sent to JVM_2 as an acknowledgement to the safe arrival at JVM_3. This startpoint is used by the endpoint on JVM_2 as a forwarding pointer to the new object location. When a method is activated on the startpoint on JVM_1, the call is still transmitted to JVM_2, where the endpoint is aware that the object has moved to JVM_3 and uses the same mechanism to forward the call.

As opposed to simple message passing, a remote method invocation is expected to return a result.[2] In a first instance, our implementation is based on call forwarding and the result is propagated back along the chain to the initial startpoint where the method call was initiated.

In such an algorithm, it is important to reduce any chain of forwarding pointers in order to reduce the cost of method invocation, but also to make the system more resilient to failures. In Fig. 3, when JVM_2 has to forward a call to JVM_3, JVM_2 knows that information on JVM_1 is out of date. Therefore, when the result is returned to JVM_1, we can also return updated information about the mobile object location. To this end, we made the remote interface implemented by endpoints different from the interface implemented by mobile objects: we return not only the "usual result", but also the new object location.

Returning updated location information at the same time as returning results may not propagate information soon enough, because processing on the server may be long. Therefore, independently, we might like to inform previous JVMs in the chain about the location of the mobile object. Since regular remote method invocation does not give any information about the method caller, we provide, as extra arguments, the stubs pointing to the JVMs involved in the chain.

In summary, a startpoint implements the same interface as a mobile object. An endpoint has a derived interface passing extra routing information, both during the forward call and during the return of a result. The RMI-stub encapsulated in the startpoint implements the same interface as the endpoint.

For the sake of illustration, let us consider the method `talk` specified in the interface `Talker` implemented by a mobile object.

```
interface Talker {
  int talk(int v, String s);
}
```

A startpoint associated with such a mobile object also implements the interface `Talker`. The endpoint of such a mobile object implements the `_Endpoint_TalkerI` interface containing a method `talk`:

```
interface _Endpoint_TalkerI {
 _int_Result talk (List from,
```

[2]In the particular case of a procedure returning a type void, no result is returned, but the method invocation terminates *after* the call has been completed remotely; for the sake of presentation, we will no longer distinguish this case from normal return of values.

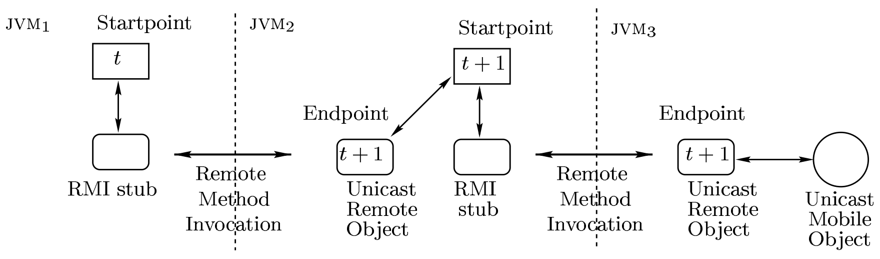


Fig. 3. Mobile object migration.

```
int _v, String _s);
}
```

The extra argument `from` is a list of RMI stubs to the JVMs that were involved in the passing of the current method call. The type `_int_Result` encapsulates an `int` as well as new routing information. We have implemented a stub compiler which takes care of generating such interfaces. It also creates the definitions of the startpoint and endpoint classes.

4.2. *Object migration*

We provide a new abstract class `UnicastMobileObject`, which encapsulates the behaviour common to all mobile objects. A mobile object must be defined as a subclass of `UnicastMobileObject`, from which two methods can be inherited:

```
protected void migrate(String url,
              Serializable state)
protected void install(Object state)
```

A mobile object can initiate its migration to another JVM, identified by a RMI-style URL, using the method `migrate`. The current object content will be serialized in conjunction with an extra argument. Upon an object's arrival, the method `install` is activated with the state argument passed to `migrate`. Both methods are defined as "protected" to guarantee that they are invoked only under the object's control.

The Java object model does not make any guarantee regarding which thread executes remote methods. Therefore, for a single object, there may be several threads executing in parallel when a request for migrating is issued by one of them. As Java does not support thread migration, it is not possible to suspend the execution of all threads in order to resume them at the destination. Instead, we allow an object to migrate when there is only one thread executing a method of this object. It is therefore the programmer's responsibility to synchronize and terminate threads currently executing in parallel, and, if necessary, to save their state in a serializable field of the object.

Mobile Objects in Java also introduce the concept of "platform", a JVM that runs mobile objects securely. A platform is a RMI `UnicastRemoteObject` which advertises its presence by binding itself with a RMI-style URL (specified at construction time) in a RMI-registry. This is such a URL which is expected as a first argument by `migrate`. Hooks are provided to perform security checks before executing objects in their sandbox [20].

4.3. *Startpoint deserialisation*

In our system, on a given platform, there is *at most* one instance of a startpoint that refers to a given mobile object. In order to preserve this invariant, each platform maintains a table of all the startpoints it knows, which is updated when startpoints are deserialised. (We use the Java method `doReadResolve` [13] to override the object returned from the stream.)

A desirable consequence of this implementation is that all objects using a specific startpoint share the benefit of the most recent routing information for that startpoint. The table of startpoints is a hash table, using a unique name given to mobile objects as a hashing key. This table uses *weak references* [11] to guarantee that startpoints do not remain accessible longer than necessary. As a result, we ensure that mobile objects may be properly garbage collected.

4.4. *Clearing routing information*

Routing information has to be cleared when it is no longer needed. Indeed, platforms run for a long period of time and host many visiting mobile objects, which leave forwarding pointers as they migrate to their next destination. We need to ensure that routing tables do not become filled with unnecessary routing information.

We have observed [22] that the task of clearing routing tables is equivalent to the distributed termination problem [25]. A forwarding endpoint is allowed to be cleared if it can be proved that no other platform will ever forward method calls to it. This may be implemented using a distributed reference counting algorithm [23,25]. In particular, RMI provides a method `Unreferenced` for remote objects which is called when there is no remote reference to this object [12]. When this method is called on an endpoint, it may be unexported, and the reference to the next startpoint in the chain may be lost. Note that this mechanism can only work if tables of startpoints contain weak references to these. Otherwise, if startpoints remain live, the RMI-stubs they contain will also remain live, which will prevent the call of the `Unreferenced` method on the associated endpoints.

5. Forwarding vs referrals

In our theoretical algorithm [22], messages are routed individually; a reply would be regarded as a separate message to be routed independently. The view that we have adopted for *Mobile Objects in Java* differs slightly because it is based on the remote method invocation paradigm: methods are invoked and are expected to produce a result. In the previous section, we showed that the result could be propagated backwards along the chain of forwarding pointers left by the mobile object.

Long chains of remote method invocations offer too little resilience to failures of intermediary nodes. Instead of forwarding a method call, an endpoint could throw an exception indicating that the mobile object has migrated. The exception could contain the new startpoint pointing at the mobile object location.

The approach consisting of throwing an exception containing a new startpoint, instead of forwarding a call, is similar to the *referral* mechanism [9] used in distributed search systems such as Whois++ [26] and LDAP [28]. It then puts the onus on the method invoker to re-try the invocation with the next location of the object; once the object has been reached, the result may then be returned to the caller *directly*. In our implementation, the startpoint is in charge of re-trying a method invocation until it becomes successful. Therefore, from the programmer's viewpoint, there is no syntactic difference between the two approaches. An option passed as argument to the stub compiler specifies whether code has to be generated for referrals or for call forwarding. In the rest of the paper, we compare the performance of the two approaches.

6. Benchmark

The scientific programming community has a tradition of adopting benchmarks to evaluate the performance of computers; for instance, the Linpack Benchmark is a numerically intensive test used to measure floating point performance. Unfortunately, we lack benchmarks specifically suited to evaluate routing algorithms for mobile objects. This may be explained by the relative novelty of the concept of mobile object, and the inexistence of widely accepted applications for mobile agents. In a previous paper [23], we observed that there was no recognised benchmark for evaluating distributed garbage collectors; therefore, we designed some synthetic benchmarks for such a type of distributed algorithms. We propose to adopt a similar approach here.

A synthetic benchmark is an abstraction of a real program, where routing of messages may have an impact on the performance of the computation. In our benchmark, we measure the cost of invoking a method on a mobile object that has changed location since the last time the method was invoked on it. In the context of the Magnitude architecture of Section 2, such a benchmark is reminiscent of the communications one may have with a mobile agent visiting several locations to perform a task.

Figure 4 summarises the "Itinerary Benchmark". An Itinerary consists of N platforms $P_0, \ldots, P_{N-1}$ to be visited by a mobile object. A platform P, not part of the itinerary, is used to initiate invocations of a method `m` on the remote mobile object. Every method invocation takes as argument a list of J platform identifiers that the mobile object has successively to migrate to; an itinerary is completed when the mobile object returns to the first platform P_0. As method `m` is invoked on the mobile object, it spawns a thread responsible for migrating the mobile object to J platforms, while method `m` terminates in parallel. On platform P, we measure the time taken to perform all method calls necessary to complete an itinerary.

Figure 5 illustrates the execution of the Itinerary benchmark over 10 platforms (5 rather heavily loaded workstations/servers each running 2 JVMs), connected by a local area network. Each method call forced the object to migrate to one new location. We ran the same benchmark using both the call forwarding and the referrals techniques. We can see that in this specific instance, referrals are on average 9% faster than call forwarding, over 200 itineraries. We should observe the abnormal duration of the first itinerary in Fig. 5: in-

Set of platforms:	$P_0,\ P_1,\ \ldots,\ P_{N-1}$
Benchmark platform:	P, *with* $P \neq P_i$
Number of jumps:	J
Number of itineraries:	C

Initial Configuration:

- *On P_0, create a mobile object o that knows of all platforms $P_0, \ldots, P_{N-1}$.*

On Benchmark platform:

- *Repeat C times:*
  - *Create a partition* $\{[n_{1_1}, \ldots, n_{1_J}], [n_{2_1}, \ldots, n_{2_J}], \ldots, [n_{k_1}, \ldots, n_{k_r}]\}$ *of integers in the range* $[1, N-1]$ *with* $N-1 = J(k-1) + r,\ r \leq J$ *;*
  - *For each subset* $[n_{i_1}, \ldots, n_{i_j}]$*:*
    - *Invoke method* `m` *on object o with arguments* $[n_{i_1}, \ldots, n_{i_j}]$*;*
  - *Invoke method* `m` *on object o with arguments* $[0]$.

Method `m` *of object o:*

- *When* `m` *is activated, with argument* $[n_{i_1}, \ldots, n_{i_j}]$*:*
  - *In a separate thread, migrate object o successively to platforms* $P_{n_{i_1}}, \ldots, P_{n_{i_j}}$ *;*
  - *Return from method* `m`.

Fig. 4. Itinerary Benchmark.

deed, it can be up to an order of magnitude slower than the others since it forces object byte-code to be loaded dynamically as the mobile object visits each platform for the first time.

In Fig. 6, we summarise our results, which we discuss now. Several variants of the Itinerary benchmark were considered.

(i) We always ran the Itinerary benchmark on 10 platforms. In one case, the platforms executed on 5 rather heavily loaded workstations/servers each running 2 JVMs) connected by a 100Mb local area network (Notation: *LAN*). In the other case, the platforms executed on 5 nodes of a cluster (Linux 2.2, 450 Mhz) with dedicated 100Mb network, with each node running 2 JVMs (Notation: *Cluster*).

(ii) The partitioning of the platforms may be deterministic or non-determinisic. In the former case, the object systematically visits platforms in the same order (Notation: *Sequential*). In the latter case, the order of platforms is decided randomly for each itinerary (Notation: *Random*).

(iii) We ran the Itinerary benchmark using both the call forwarding (Notation: *CF*) and the referrals techniques (Notation: *Ref*).

(iv) When a mobile object migrates to successive locations, its new position can be acknowledged to all its previous locations (Notation: *Eager Acknowledgement*), or to its directly previous location only (Notation: *One Acknowledgement*).

In order to reduce some of the non-deterministic nature of the benchmark, we have introduced a delay between each method call to the mobile object, which gave time to the object to migrate to its location. Such a delay is not included in the results.

In the first table of Fig. 6, eager acknowledgement of object migration resulted in methods calls to be forwarded at most once. This is confirmed by the average duration of a method call, which does not incur any significant variation as J, the number of migrations associated with a method call, increases. We also observe that there is no significant difference between sequential and random itineraries. Finally, the referrals technique appears to be marginally more efficient than call forwarding.

In the second table of Fig. 6, acknowledgements of object position is back-propagated to the object's previous location only. Therefore, as we increase J, the number of platforms that the mobile object has to migrate to for each method call, we observe that method calls have to be forwarded further. Again, we do not observe any significant difference between sequential and random itineraries. However, the referrals technique becomes significantly more efficient than call forwarding: its efficiency is in the range [11%–19%] for a LAN, whereas its in the range [6%–11%] for a cluster.

Instrumenting the Itinerary benchmark turned out to be more difficult than anticipated. Indeed, many ele-

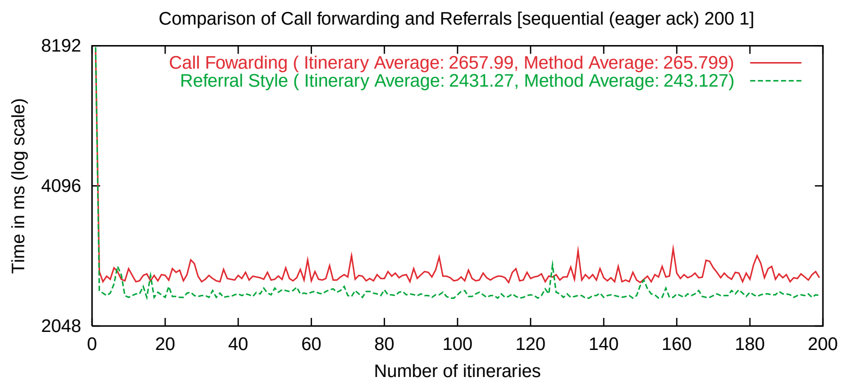


Fig. 5. An illustration of call forwarding vs referrals (LAN).

Eager Acknowledgement $(N = 10, C = 200)$

		LAN						Cluster					
		Sequential			Random			Sequential			Random		
J	Method calls/Itinerary	CF	Ref	%	CF	Ref	%	CF	Ref	%	CF	Ref	%
1	10	266	243	9%	263	245	7%	254	243	5%	255	244	5%
2	5	260	244	7%	260	245	6%	258	246	5%	259	246	5%
3	4	262	244	7%	261	245	7%	257	246	4%	257	251	2%
4	3	257	245	5%	257	249	3%	257	256	7%	260	255	2%
5	2	248	248	0%	256	256	0%	257	261	2%	256	257	0%

One Acknowledgement $(N = 10, C = 200)$

		LAN						Cluster					
		Sequential			Random			Sequential			Random		
J	Method calls/Itinerary	CF	Ref	%	CF	Ref	%	CF	Ref	%	CF	Ref	%
1	10	— *Identical to Eager Acknowledgement* —											
2	5	299	264	11%	299	263	12%	275	258	6%	278	258	7%
3	4	323	278	14%	330	280	15%	295	271	8%	298	269	10%
4	3	343	289	16%	348	293	16%	312	279	11%	312	280	10%
5	2	329	287	13%	364	295	19%	315	288	9%	312	288	8%

Fig. 6. Average duration of a method call to a mobile object.

ments, not in our control, interact with our implementation. In particular, platform to platform communications were implemented with Java RMI, which uses Birrel's distributed garbage collector [4]. Such a distributed GC introduces synchronisations every time a stub is communicated by a remote method invocation; in particular, such synchronisations occur in the benchmark when an object migration is acknowledged, or when stubs are piggybacked. An alternative would be to use another algorithm [23] which does not introduce such synchronisations. Our rationale for comparing sequential and random itineraries was to test whether a cost was incurred because new connections needed to be opened. Java RMI hides the implementation details in a totally opaque manner, and we have no control over the management of these resources in our implementation.

Discussion

Call forwarding requires two interventions of each intermediary platform for forwarding the call and the result, whereas referrals require only one such intervention. We believe that this element is the principal explanation for the superior performance of referrals in the presence of heavily loaded platforms (as in our LAN). We anticipate that such a configuration is similar to the environment in which mobile agents are likely to be deployed (cf. Section 2).

At the beginning of our investigation, we debated whether referrals would be penalised by having to open new connections between the benchmark platform and itinerary platforms. In all likelihood, such connections had to be opened for distributed GC purpose in both variants of the algorithm, and therefore no significant change of performance could be attributed to this aspect. Tools to instrument resources used within the JVM would be extremely valuable in this context.

7. Related work and conclusion

We have presented *Mobile Objects in Java* a library able to route method invocations to mobile objects. We have discussed two ways of forwarding calls, namely call forwarding and referrals; the latter turned out to be more efficient in our benchmark. There is a third method where the caller explicitly passes a reference to itself, which is used by the callee to return the result. Such a method discussed in [8,21] allows the result to "short-cut" the chain of forwarding calls. A more extensive study is required to investigate the performance of these three methods (as well as the home agent approach) in various scenarios.

Mobile Objects in Java is an integral part of a mobile agent system that we use to support mobile users in the Magnitude project [24]. From a software engineering viewpoint, such a library provides a separation of concern between higher-level interactions and message routing. We are adopting such a communication model in three different circumstances.

(i) User-driven communication to their mobile agents;
(ii) Return of results from a mobile agent to a mobile personal digital assistant;
(iii) Communications between mobile agents.

There are a number of other systems that support mobile computations, but they adopt a different philosophy. Emerald [15] supports migration of an object, including threads running in parallel. In Kali Scheme [6], continuations may be migrated between address spaces. None of them provides the transparent routing of messages, as described in this paper. Other approaches rely on a stationary entity to support communications between mobile objects, including Aglets [16], Nomadic Pict [29], April [18] and the InterAgent Communication Library [19]. Jumping Beans [2] is a commercial product offering support for mobile applications, but requires a server to be visited by mobile agents during each agent migration. Stationary and central locations put an extra burden on the infrastructure which we wanted to avoid in our implementation.

Our investigation has highlighted a number of difficulties concerning the evaluation of algorithms for mobile agents.

(i) In high-level implementations such as ours, in particular above Java RMI, the lack of tools to instrument low-level resources (connections, distributed garbage collection) makes it somewhat difficult to explain observed behaviours.
(ii) The absence of widely recognised benchmarks does not ease comparison with other authors.
(iii) In mobile computing, social human behaviours dictate the patterns of physical mobility; these can be extensively used in simulations. Because we lack widely accepted applications of mobile agents, we also lack accepted models of their mobility.

It is this specific problem that Huet [10] addresses by looking at a formal modelisation of routing algorithms as stochastic processes. In particular, he compares a centralised forwarder with distributed forwarding pointers. From the slides that were accessible to us, we were enable to establish the patterns of mobility he adopted, and whether call forwarding or referrals were considered. A challenging issue is to define simulations that are refined enough to take into account other activities such as distributed garbage collection, which itself also lacks recognised benchmarks.

In the future, we wish to investigate strategies for propagating information about object's locations independently of remote method invocation. Such a study will have to consider new benchmarks, ideally derived from real applications, and should also include alternative routing algorithms. Furthermore, other requirements and their implications on performance need to be investigated, such as security and robustness of directory services.

Acknowledgements

David De Roure pointed out an analogy between the directory service described in this paper and the notion of referrals used in distributed search systems and in "query routing". Thanks to Omer Rana for discussions on their agent-based Digital Library system, and to Danius Michaelides for his comments on the paper. This research is funded in part by EPSRC and QuinetiQ project "Magnitude" reference (GR/N35816).

References

[1] S. Adams and D. DeRoure, A Simulator for an Amorphous Computer, in: *Proceedings of the 12th European Simulation Multiconference* (*ESM'98*), Manchester, UK, June 1998.
[2] Ad Astra, Jumping beans, Technical report, White Paper, 1999, http://www.JumpingBeans.com/.
[3] A.D. Birrell and B.J. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems* **2**(1) (February 1984), 39–59.

[4] A. Birrell, G. Nelson, S. Owicki and E. Wobber, Network Objects, Technical Report 115, Digital Systems Research Center, February 1994.

[5] L. Cardelli, Abstractions for Mobile Computation, in: *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, J. Vitek and C. Jensen, eds, Vol. 1603 of *Lecture Notes in Computer Science*, 1999.

[6] H. Cejtin, S. Jagannathan and R. Kelsey, Higher-order distributed objects, *ACM Transactions on Programming Languages and Systems* **17**(5) (September 1995), 704–739.

[7] J. Dale and F.G. McCabe, Agent Management Support for Mobility, Fipa'98 draft specification, Fujitsu Laboratories of America, 1998.

[8] I. Foster, C. Kesselman and S. Tuecke, The Nexus Approach to Integrating Multithreading and Communication, *Journal of Parallel and Distributed Computing* **37** (1996), 70–82.

[9] N. Gibbins and W. Hall, Scalability issues for query routing service discovery. in: *Proceedings of the Second Workshop on Infrastructure for Agents, MAS and Scalable MAS*, May 2001.

[10] F. Huet, Distribution and localisation, http://www.irit.fr/ACTIVITES/PLASMA/PRO-Toulouse2001/LesPropositions/LesTransparents/pro-huet.pdf.

[11] *Java Reference Objects*, http://java.sun.com/j2se/1.3/docs/guide/refobs/.

[12] *Java Remote Method Invocation Specification*, November 1996.

[13] *Java Object Serialization Specification*, November 1998.

[14] D.B. Johnson and C. Perkins, Mobility Support in IPv6, Internet draft, IETF Mobile IP Working Group, 1999. draft-ietf-mobileip-ipv6-09.txt.

[15] E. Jul, Migration of light-weight processes in Emerald, *Operating Systems Technical Committee Newsletter* **3**(1) (1989), 25–30.

[16] D.B. Lange and M. Ishima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, 1998.

[17] General Magic, Telescript Technology: Mobile Agents, 1996.

[18] F.G. McCabe and K.L. Clark, APRIL – Agent Process Interaction Language, in: *Proc. of ECAI'94 Workshop on Agent Theories, Architectures and Languages*, Springer-Verlag, 1995.

[19] F.H. McCabe, InterAgent Communications Reference manual, Technical report, Fujitsu Laboratories of America, 1999.

[20] G. McGraw and E.W. Felten, *Securing Java*, Wiley, 1999.

[21] D. Michaelides, L. Moreau and D. DeRoure, A Uniform Approach to Programming the World Wide Web, *Computer Systems Science and Engineering* **14**(2) (1999), 69–91.

[22] L. Moreau, Distributed Directory Service and Message Router for Mobile Agents, *Science of Computer Programming* **39**(2–3) (2001), 249–272.

[23] L. Moreau, Tree Rerooting in Distributed Garbage Collection: Implementation and Performance Evaluation, *Higher-Order and Symbolic Computation*, To appear.

[24] L. Moreau, D. De Roure, W. Hall and N. Jennings, MAGNITUDE: Mobile AGents Negotiating for ITinerant Users in the Distributed Enterprise, http://www.ecs.soton.ac.uk/~lavm/magnitude/, 2001.

[25] G. Tel and F. Mattern, The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes, *ACM Transactions on Programming Languages and Systems* **15**(1) (January 1993), 1–35.

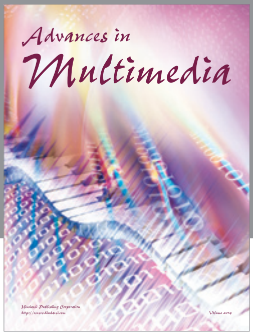
[26] C. Weider, J. Fullton and S. Spero, Architecture of Whois++ Index Service, Request for comments 1913, Internet Engineering Task Force, 1996.

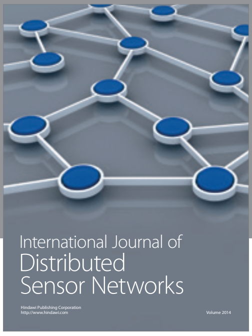
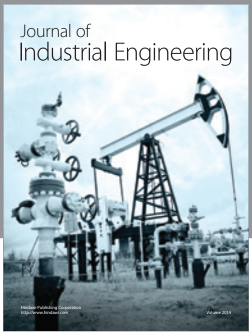
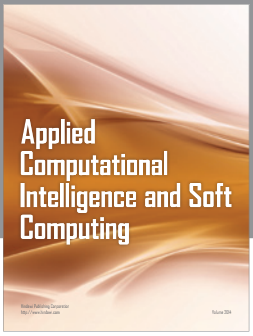
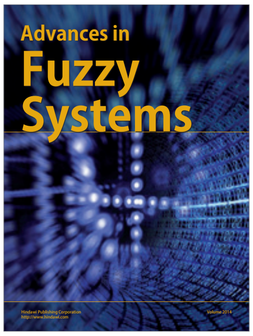
[27] M. Weiser, Some Computer Science Problems in Ubiquitous Computing, *Communications of the ACM* **36**(7) (July 1993), 74–84.

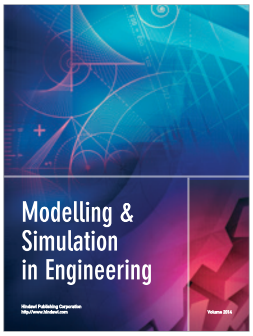
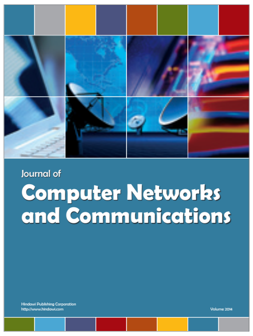
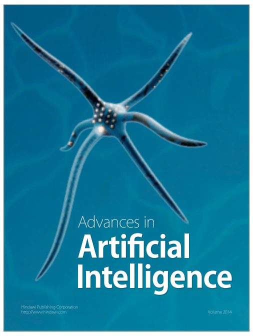
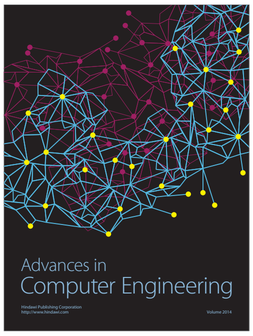
[28] M. Whalh, T. Howes and S. Kille, Light Weight Directory Access Protocol (v3), Request for comments 2251, Internet Engineering Task Force, 1997.

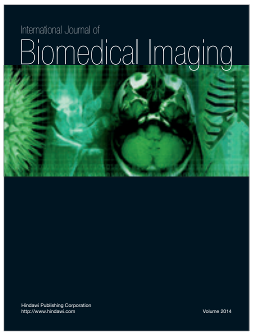
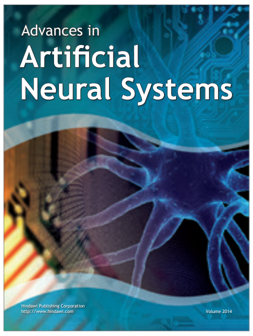
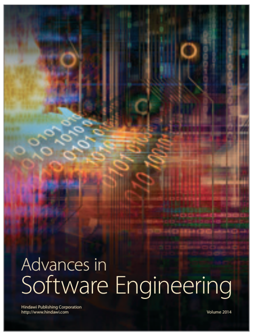
[29] P. Wojciechowski and P. Sewell, Nomadic Pict: Language and Infrastructure Design for Mobile Agents, in: *First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents* (*ASA/MA'99*), October 1999.

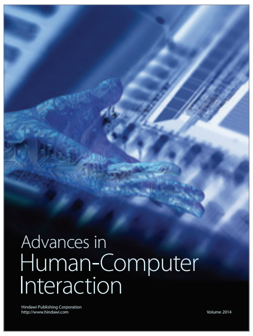
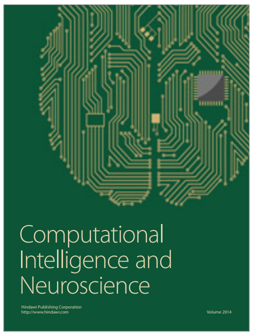
[30] Y. Yang, O.F. Rana, C. Georgousopoulos, D.W. Walker and R.D. Williams, Mobile agents and the sara digital library, in: *Proceedings of IEEE Advances in Digital Libraries 2000*, Washington, DC, May 2000, pp. 71–77.


Advances in
Multimedia

The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems


Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering