Jada: coordination and communication for Java agents

Paolo Ciancarini & Davide Rossi
E-mail: {cianca,rossi}@cs.unibo.it

Dipartimento di Scienze dell'Informazione
University of Bologna - Italy

Abstract. In this paper we are going to analyze mobile code issues in the perspective of Object Oriented systems in which thread migration is not supported. This means that both objects' code and data can be transmitted from a place to another but not the current execution state (if any) associated to the object. This is the case with the Java language which is often used in the Word Wide Web for developing applets which are little applications downloaded on the fly and executed in the client machine. While this mechanism is quite useful for enhancing HTML documents with sound and animation, we think that this technology can give its best in the field of distributed-cooperative work, both in the perspective of Internet and Intranet connectivity. Java is indeed a concurrent, multithreaded language, but it offers little help for distributed programming. Thus, we introduce Jada, a coordination toolkit for Java where coordination among either concurrent threads or distributed Java objects is achieved via shared object spaces. By exchanging objects through tuple spaces, Java programs and applets can exchange data or synchronize their actions over a single host, a LAN, or even the Internet.

The access to an object space is performed using a set of methods of the ObjectSpace class. Such operations are out (to put an object in the object space), in and read (to get or to read associatively an object from the object space), and others, mostly inspired by the Linda language.

Jada does not extends the syntax of Java because it is a set of classes. We show how it changes the way we design multiuser, distributed applications (such as the ones based on the WWW) allowing easy interactions between software components and agents.

Under this perspective we can outline a system of any scale which uses the dynamic linking capability of Java to distribute the code and the coordination facility of Jada to handle distributed entities inter-relations.

1 Introduction

Java is a new language and programming environment developed at Sun Microsystems and its main application field is in conjunction with the Word Wide Web (many WWW browsers can indeed download HTML documents which integrate Java code) so the natural Java environment is a distributed system.

Although Java is multithreaded and allows remote dynamic code loading, it is not a language for distributed programming. The only facility offered by
the language for supporting distributed processing is a sockets library. However, programming distributed systems using sockets is notoriously boring, difficult and error-prone. An alternative strategy consists of adding to Java high-level coordination mechanisms: this is the theme of this paper.

This document has the following structure: in Sect. 2 we discuss how dynamic linking can be exploited to allow code mobility in an Object Oriented environment; in Sect. 4 we shortly recall Java and we show how it handles mobile code; in Sect. 3 we give an small overview of coordination and coordination languages; in Sect. 5 we introduce Jada (Java+Linda); in Sect. 6 we discuss how we implemented Jada; in Sect. 7 we discuss how Jada can be used to solve well known coordination problems; in Sect. 8 we compare our system with other similar systems. Sect. 9 concludes the paper.

2 Mobile Code

Stand-alone, centralized computers are rapidly replaced and integrated by networked, distributed computer systems. The most widespread computer model that takes advantage of these systems is Client/Server: on behalf of the user, the Client computer requests services from a remote Server; the Server execute the requested computations and send a reply back to the Client. The advantage of this model is that users with low cost machines can access complex services, and organizations can optimize companywide distribution of services to minimize cost and maximize efficiency. Now the networks are expanding to global, worldwide scale: each computer system is an access point to information from all over the world. People using the Internet are connecting from their sites (possibly their homes) to servers located anywhere in the network, crossing several types of physical networking (satellite, optical fiber, wireless, etc.), each with a very different type of bandwidth and latency. The overall performance and behavior are not that of local network or leased lines. In some cases people travel with their laptops or hand-held machines and connects them to the network just from time to time, which implies network connections can be geographically moved and turned on and off too (this kind of systems are usually referred to as nomadic computing). In this scenario the Client/Server model suffers some of its limitations: it is impossible, for example, handling applications in which a client wants for the server to continue computations even though the client has been disconnected from the network (or even turned off), and get the results when it is back on-line, or applications where the network bandwidth or latency make the application impossible to run with satisfactory response time. A solution to those problems is to use mobile code. In this model the code that have to be executed to satisfy a request is sent over the network to execute onto the remote place. This model easily overcomes Client/Server limitations about the client being off-line and latency and bandwidth issues. This is often the case with Clients that need large amount of data from a Server to compute some kind of operation that results in a small amount of data; we can ship the code to compute the operation to the Server and get the reply later, without requiring
any further network traffic.

2.1 Security issues

The use of mobile code introduces security issues: mechanism like authentication, authorization and privacy are required with this model in open distributed environments.

To enforce security the underlying system must be safe: we need to be able to implement security mechanisms and precludes a program to defeat them. A system where pointer forging and arithmetics on pointers are allowed, or where filling a memory buffer with code data and jumping to it is possible will never be a safe system.

Thus the use of mobile code in open distributed environments requires a language and an environment that is safe; it will then be possible to introduce the various security mechanisms.

2.2 Dynamic Linking: code on the move

Once the mobile code has been sent over a network, the receiving host must run it. Different approaches are possible: the receiving host could wrap this code in its own address and data space and run it as a separate process; or it could use its entry point just like any other function’s entry point of the running program and make a jump to it. In a Object Oriented environment we can think about a whole object to be received (with its transitive closure or part of it); a reference could be made to that object which could be then accessed just like any other object in the system. We can make this mechanism transparent with the use of dynamic linking: references to objects are solved at run-time, allowing dynamic system configuration/programming. New classes can be added to the running program allowing objects downloaded from a network to be integrated with the executing system. Thus dynamic linking can be exploited to elegantly implement mobile code systems, mostly in Object Oriented environments.

3 Coordination

We can think of a distributed program as a agent world in which agents are spatially dispersed and act autonomously (this image fits quite well the distributed objects model) [Tol94]. Processes (sequence of actions performed by an agent) are the executions of methods and communications are message sent to objects. Synchronization actions (starting, blocking and unblocking, terminating) cover the remaining mechanisms in object invocation. A service occurs when an object invokes methods of another object (the invoker is the service-user and the invoked object is the provider).

Coordination is about the actions for synchronization, communications and service usage and provision. Computation is about the actions of processes.
3.1 Coordination Languages

The coordination languages have their focus on separation between coordination and computation, not meaning that they are independent or dual concepts, but meaning that they are orthogonal: they are two dimensions under which an agent world can be examined [CG92]. Most of the coordination languages are based on shared data spaces, like Linda [Gel85] (where focus is given on communication issues) and Laura [Tol92] (where focus is given on services), and a set of operators to access the data in the spaces.

4 Java

Java is an object oriented language whose syntactical structure resembles C++. However, it inherits a better object-oriented semantics from SmallTalk and Objective C [AG96].

The language model is somehow limited but its strongest point is its great simplicity.

Java programs are made of objects. Java objects are instances of classes. A class can inherit from a base class (just one, multiple inheritance is not supported) to extend the behavior of that base class.

Java programs are compiled into a binary format [Gos95] that can be executed on many platforms without recompilation using a “bytecode interpreter”, also known as Java Virtual Machine (JavaVM). The bytecode is linked on the fly when loaded to be run [Gos95]. All the dependencies in the byte-code are symbolic and are resolved only at load time. The loading of a class code happens only when it is required. Furthermore the user can extend the manner in which the Java Virtual Machine dynamically loads classes, creating, for example, a network class loader in order to load a class code from a remote server, implementing code mobility.

Thus, the JavaVM contains mechanisms to check the code it has to run, in order to avoid viruses and security violations.

Java is architecture neutral: every machine-dependent code is not part of the language itself but of its standard library. Java libraries are collections of classes called “packages”. We can think of Java as an environment built by the language itself and its standard packages. Several aspects of the language depend indeed from the standard packages as defined by Java designers. For instance, multithreading is native in Java, but there is no way to create a thread other than using a method of the Thread class.

In the standard Java environment a set of packages is provided; those packages are:

- java.lang this package is automatically imported when compiling. It contains the definitions for the root class Object and for the meta-class Class, plus threads, exceptions, etc...
- java.io contains the classes for accessing files.
- java.net contains the classes for accessing the network.
- **java.util** contains utility classes such as containers, date and time managers, and so on.
- **java.awt** contains the classes for the Abstract Windowing Toolkit, which defines basic graphics interface components in a form that is independent from the actual windowing library the implementation uses (eg. Windows API for Windows and Motif for Unix).

### 4.1 Java and the Internet

HotJava, a WWW browser, has been the first Internet application of Java. What made it different from other browsers is its ability to run Java code “embedded” within HTML documents. This way HTML documents become “active”, namely it is easy to put animation in HTML pages. Moreover, the same mechanism can be used to extend a browser in a natural way, implementing, for example, editors, spreadsheet, and groupware applications made of distributed objects. The most important consequence of having Java-enabled browsers like HotJava is that a Java application can be “spread” around the network. The simplest way to take advantage of this feature is to think about a Java application just like a document. Watching a Java application means let it run in your host, possibly integrated within your browser. The Java way to obtain this is to build each piece of Java code to be run in a browser as an “applet”. It results that, extended this way, the Word Wide Web can be seen as a geographically scaled, ubiquitous, mobile objects environment.

An **applet** is the byte-code result of a compilation of a class that extends the class **Applet** contained in the **java.applet** package. This means that each applet inherits a standard behaviour from **Applet** and has to comply with a set of conventions which let them run within a Java-compatible browser.

### 4.2 Distributed Java applications

Besides the basic classes shown above, Java suffers the lack of mechanisms. The only way we have for two applets/applications to communicate is to use sockets to establish a bidirectional point-to-point link. Moreover, sockets require the client and server to engage in applications-level protocols to encode and decode messages for exchange, and the design of such protocols is cumbersome and can be error-prone, also: applets have security restrictions that limit the usability of sockets. Things get even worse when we have the need for the objects to cooperate in complex fashions.

In a first attempt to seriously overcome this kind of limitations a distributed object model for Java has been proposed: the **remote method invocation** (RMI). RMI uses the well known CORBA [BN95] approach which, while allows easy distributed objects basic interactions, doesn’t allows any kind of mobility.

Our approach is a step further: we want add to Java coordination capabilities in order to allow any kind of complex interactions between potentially remote objects.
5 A toolkit for programming distributed Java applications

We solve the coordination problem for Java using a well known approach, namely adding Java a minimal set of coordination primitives: in particular, we add operations to access Linda-like multiple object spaces [CG92, Cia94].

We designed Jada aiming to simplicity rather than performance. Jada, like Linda, is a minimalist coordination language. Differently from other Linda-like implementations, which usually include a preprocessor necessary because Linda slightly changes (i.e. it constrains) the host language syntax, Jada is based on a set of classes to be used to access an object space, allowing the users to keep their standard Java development tools.

5.1 Object Spaces

An ObjectSpace is an objects container with a set of methods for accessing its contents. out is the method to put objects in the ObjectSpace, in and read are the methods to get objects from the ObjectSpace (Jada allows many kind of these basic operations).

To create an object space we write:

ObjectSpace myObjectSpace = new ObjectSpace();

To put an object in the object space we can write:

myObjectSpace.out(new String("foo"));

To access the ObjectSpace for getting objects we use an associative mechanism: when the user calls the in method he have to pass it an object which is used as a matching pattern. The object the in method will return (if any) is an object from the object space that matches the given matching pattern (the same applies to the read method; the difference between read and in is that in removes the returned object from the object space).

To allow enough flexibility to the matching operations we introduce the concepts of formal and actual objects: a formal object is an instance of the Class class (the meta-class used by Java). Any other object is an actual object.

The Jada matching rules are:

- **actual-actual**: two actual items are matching if
  - they implement the JadaItem interface: the method matchesItem applied to the object in the object space, passing as parameter the other object, returns true.
  - they do not implement the JadaItem interface: the method equals applied to the object in the object space, passing as parameter the other object, returns true.

- **actual-formal**: a formal item matches any object which is an instance of it.

- **formal-formal**: two formal items are matching if they represent the same class.
Please note: in the current version of Jada the matching mechanism is object based and not object oriented; this means that inheritance is not applied when checking for two objects to be of the same type: the matching is performed just on object which are directly instance of the same class; we are planning to extend this matching mechanism to inheritance in the future.

These matching rules are quite simple yet very powerful; but are sometimes not enough expressive to match the requests of the user. Thus we introduce the Tuple class, an object container class with an extended matching mechanism which is more expressive.

5.2 Tuples

In Jada a tuple is a set of objects (also referred as items) and it is represented by the jada.Tuple class.

This is an example of Jada tuple:

```java
Tuple my_tuple=new Tuple(new Integer(10),"test");
```

Such a tuple includes two items (we say that its cardinality is two); the first item is an Integer object, the second one is a String object. We define actual and formal items within a tuple the same way we defined them for Jada.

To use associative object space access with tuples we can use the tuples’ matching mechanism:

two tuples a and b are matching if they have the same cardinality and each item of a matches the corresponding item of b. The usual Jada mechanism is used to check if the items are matching.

Thus, the tuple

```java
Tuple a= new Tuple(new Integer(10),"test");
```

matches the tuple:

```java
Tuple b= new Tuple(new Integer(10),new String().getClass());
```

Note that to exchange a tuple (and generally any kind of object) two threads do not need to perform a pair of out and read operations at the same time (Jada does not need rendez-vous communication). In fact, suppose the threads ta and tb want to exchange a message: ta will put a message inside the object space, tb will read the message from the object space. If ta performs the out operation before tb performs the read operation it does not have to wait for tb: it simply continues its execution, the tuple is now stored into the object space. When tb performs the read operation it will be able to read it.

Instead, if tb performs the read operation before ta performs the out operation, tb will be blocked until an object that satisfy the read request will become available (i.e. until ta performs the out operation).

The in and read methods are indeed blocking. If you want the thread not to be blocked when a matching object for the in and read operations is not available you can use the in.nb and read.nb methods: they access the object space the same way as in and read but if no matching object is available they
simply return null (a more sophisticated flavor of in and read that aborts after a time-out is available too).

5.3 ObjectServer and ObjectClient

To allow remote access to an object space, the jada.net.ObjectServer and jada.net.ObjectClient classes are provided. We used a simple client/server architecture to manage the object spaces; in fact, each object space is a shared remote resource accessed through an object space server.

The object space server is addressed using the IP address of the host it runs on and with its own port number (as usual with socket connections). This way we can run (almost) as many object space servers as we like in a network, so that applications can independently operate on several, distributed object spaces. ObjectServer is a multithreaded server class which translates requests received from the ObjectClient class in calls to the methods of the ObjectSpace class.

In fact, both ObjectServer and ObjectClient are based on ObjectSpace. ObjectServer and ObjectClient communicate using sockets.

ObjectServer uses ObjectSpace to perform the requested operations.

The ObjectClient class extends ObjectSpace changing its internals but keeping its interface and behavior (apart from some new constructor and few new methods). Thus, a ObjectClient object is used just like a ObjectSpace one, except that it provides access to a remote object space which can run in any host of the network.

What ObjectClient does is to interface with a remote ObjectServer object (which holds the real object space) and requests it to perform the in, read and out operations and (eventually) to return the result.

6 Some details on the implementation

Jada is implemented as a set of classes that allow either Java threads or Java applications to access associatively a shared object space using a small set of Linda-like operations.

An object space can be either local to the application or applet (ObjectSpace, to be shared between threads) or remote (ObjectClient, to be shared between applications or applets) in order to build a distributed application.

In the latter case an object space server (ObjectServer) must be running in a host of the network.

In Fig. 1 we graphically outline how an application can access a remote tuple space.

6.1 Serialization

To handle object space access we need our classes to provide:

- a method to dump the object contents to a byte stream, in order to be able to send a tuple across a network or to save its state to a file. (dumpItem).
Fig. 1. Client/Server relationships in Jada
- a method to restore a tuple given a set of bytes read from a byte stream. (buildItem).

These two tasks are operations we usually need when dealing with distributed objects and they are often referred as serialization operations. Jada allows two serialization mechanism:

- serialization via JadaItem. If the object we want to serialize implements JadaItem he must have defined the code for the dumpItem and buildItem methods which are used respectively to dump the object to a stream and to build an object from a stream.
- serialization via JadaSerializer. If the object we want to serialize does not implement JadaItem there must be a JadaSerializer object defined for such object type that as been passed to the Jada runtime system using the addSerializer method of the JadaSystem class. This must be done on both client and server side. JadaSerializer is an interface that must be implemented by classes which are used to serialize objects. These classes must then have defined the code for the dump and restore methods, which are used to serialize the object. When the Jada system is initialized the default serializer for the Integer, Float and String classes are provided to the system.

When, in the future, some standard serialization operation will be provided as part of the Java environment we plan to add these operations as a third serialization mechanism.

Serialization must, of course, also be supported by tuples’ items (we indeed need to know how to dump/restore them in order to dump/restore a tuple).

The Tuple class itself implements the JadaItem interface so we can use a Tuple object as a field of a tuple.

6.2 ObjectSpace

An ObjectSpace object in Jada is an object container which offers a set of thread-safe access methods. Thread-safe means that accessing an ObjectSpace from different threads at the same time is safe since monitors are used to handle requests avoiding problems with critical regions.

The methods are the usual in, read and out (along with their multiple, non-blocking and time out-aware variants). All these methods are actually just wrappers for the doInRead and doOut methods, which are the real “engines” for this class. All the synchronization and repository management code is part of these methods. This allows to easily redefine ObjectSpace behaviour by extending its class just like we did with ObjectClient. For example the doOut method of ObjectSpace takes care of putting the specified object in the object space or use it to reply pending in or read requests. The ObjectClient’s version of doOut, instead, sends the tuples to an object server and asks it to manage the storing/replying. The same applies for ObjectClient’s version of doInRead.
We need indeed just to change these two methods to deal with a remote tuple manager (which is a ObjectServer object) and use socket connections to talk with it.

6.3 ObjectServer

As stated above an ObjectServer object is used by ObjectClient objects to access a (possibly remote) object space. An ObjectServer has an ObjectSpace object in it which is used to manage the object space. Each time an ObjectClient perform a request the ObjectServer spawns a new thread to perform the requested operation. From the architectural point of view we can think about ObjectServer/ObjectClient like a stub/proxy system.

If we analyze the behavior of a Jada program we can distinguish two main cases:

- the use of an ObjectSpace shared among concurrent threads belonging to the same application, a situation symbolically depicted in 2.

![Diagram](image)

**Fig. 2.** Local threads accessing an object space

Each shape in such a picture represents a thread running within an application. Thread1, Thread2, and Thread3 are interacting with the object space (using ObjectSpace methods). The ObjectSpace object manages their access and takes care of handling critical regions managing. Note that some of the thread may have blocked calling a method to perform a blocking in or read operation.

- the use of an ObjectSpace shared among concurrent threads belonging to different applications. In this case we have to use an ObjectServer/ObjectClient architecture to access the shared space, as we shown in 3. Thread1 and Thread2 are now part of an application while Thread3 is part of a different one. The former are using the same ObjectClient object to
access the object space, while the latter uses a private one. When a thread performs a request calling a method of the **ObjectClient** object, the request is forwarded to the **ObjectServer** object, using a network connection, which runs a thread to handle it. Each thread run by the object server then corresponds to a remote thread which performed a request (this is actually not always true because of the optimizations we did to the package, this general concept, however, is more easy to understand).

We can see how the former situation is automatically replicated in the latter one inside the application which runs the **ObjectServer**. where the threads that are created to handle remote requests behave just like the threads in a local **ObjectSpace**, giving evidence to the implicit scalability of Jada’s architecture.
7 Programming with Jada

Using Jada to allow threads to exchange data is quite an easy task but we can do even better. We can use Jada to coordinate threads, thus easily solving many well-known coordination problems:

- server/client model: a server read requests in form of objects from the object space. A client performs a request putting an object in the object space.
- master/worker model: the master put jobs in form of objects in the object space, then reads results from it. Workers read jobs from the object space, execute their task, and then put the result into the object space.
- producer/consumer model: we decouple a producer and a consumer using the object space as temporary repository for data to be exchanged.
- mutual exclusion: we can use an object as a “token” to enter a critical section: any thread need to get such a token to enter the section. Upon exiting the section the thread releases the token (i.e. the token object can be put back into the object space).
- message passing, either synchronous or asynchronous: a sender puts the message into the object space (and in the synchronous case waits for an ack object). The receiver reads the message from the object space (and in the synchronous case puts an ack object in the object space).

Many other concurrency problems can also be easily solved using shared object spaces [CG90].

We are using Jada as coordination kernel for implementing more complex Internet languages and architectures. In particular, we are developing on top of Jada the Shade/Java agent based coordination language [CCR96], and PageSpace [CKTV96], a coordination architecture for using the WWW as middleware for building cooperative applications (eg. groupware, electronic commerce) over the Internet.

8 Related work

Jada has been developed in the context of the PageSpace project [CKTV96]. The main idea at the basis of PageSpace is to exploit coordination technology to enhance the WWW middleware. Other projects pursue a similar goal with different approaches.

In fact, although it explicitly refers to Linda, the WWWindla approach [GNSP94] is quite different from what we have described here. Since the integration between WWW browsers and their helper applications is extremely rudimental (it is only possible to activate the external application and to deliver the data to be displayed), the WWWindla research team designed a flexible, modular WWW browser architecture based on the Linda programming language. Several independent tools, each implementing a different part of the whole WWW browser, are activated according to needs, sharing screen space, HTTP responses, and
user interaction. This allows for a highly modular architecture, where new services and tools can be added without modifications in a homogeneous framework. In order to allow cooperation among helper modules, these all make use of a shared tuple space. Tuples allow to upgrade the simple “activate with these data” paradigm of browsers’ helper applications to a more complete coordination protocol. For instance, WWWinda has been used to coordinate a distributed digital orchestra, in which several browsers simulating musical instruments (possibly running on different machines) extract from the shared tuple space the tune to be played note by note. No instrument is aware of how many other instruments are present, and new ones can be added on the fly, even in the middle of a performance.

All features of WWWinda, that we define a “client-side coordination toolkit” are easily implementable in Jada.

Instead, the WU Linda Toolkit [Sch95] is a simple interface between a WWW browser and a WWW server implementing a tuple space service. Access to the shared tuple space is provided to clients: users can fill in an HTML form with the appropriate Linda command to interact with the tuple space. The main application on show is a disc-load viewer that allows a first glance check of current disk usage of the computers of a cluster of workstations. Each workstation hosts a WU Linda client which posts tuples in the shared tuple space describing the current load of the disks it controls. These tuples are then collected and rendered in a user-friendly way via HTML to any browser querying the application.

Again, all features of WU Linda, that we define a “server-side coordination toolkit” are easily implementable in Jada.

Jada applets are very similar to Oblets, or “distributed active objects” [BN96]. An oblet is a program written in Obliq and executed on a browser. Each oblet can use high-level primitives to communicate with other oblets running on possibly remote browsers.

The Bauhaus “Turingware Web” designed in Yale (the homeland of Linda) is similar at least in spirit to the WU Linda toolkit [Hup96]. The main idea consists of using a standard browser to access a Bauhaus server. Bauhaus is a coordination language based on nested multiple tuple spaces (multisets) which can be used in this case for both controlling the hierarchical structure of the pages of a web site, and for associating agents and their activities to the pages themselves. For instance, one attribute of a page could be the list of users “acting” in such a page, who are displayed by a graphic icon and can interact using some ad-hoc cooperation services.

This application of coordination technology to the WWW is based on a language richer than Linda. We are also investigating similar extensions, based on multiple tuple spaces, in a coordination language called Shade, whose kernel is currently written in Jada [CCR96].

9 Conclusion

Jada provides:
- coordination inside the Object Oriented Programming framework: no syntax extension for Java, just a set of classes. Each data type used by Jada is a Java object.
- dynamic tuple creation: being an object, a tuple can be created with new and many different constructors are provided in order to build a tuple from a string, an array of object or, simply, from a set of arguments.
- multithreading support: Jada is multithreading aware: different threads can access the same object space; blocking request are managed at thread-level.
- open systems support: at any time threads or applications can perform operation on an object space.
- multiple objects associative access: we provided modified flavors for the in and read requests in order to allow the use of multiple matching objects.

Acknowledgments. We thank Robert Tolksdorf for discussions and suggestions. We thank for support the EU Open LTR project Pagespace n. 20179. We have been partially supported also by Italian CNR - "Progetto Coordinato Programmazione Logica".

References


