Java Does not Distribute

Gerald Brose, Klaus–Peter Löhr, André Spiegel

{brose,lohr,spiegel}@inf.fu-berlin.de

Technical Report B–97–07
October 1997

Abstract

Java is commonly considered the ideal language for implementing software for the Internet. A closer look, however, reveals that distributed programming is poorly supported in Java. This is because the very design of the language rules out distribution–transparent remote invocation. It is shown that Sun's technology for distributed Java programming, RMI, makes things worse by allowing two different invocation semantics to hide behind an object variable. The consequences of using CORBA instead of RMI are investigated. Various options for changing either RMI or Java itself are considered, so that language platforms supporting a high degree of distribution transparency could be built.

Freie Universität Berlin
Institut für Informatik
Takustraße 9
D–14195 Berlin, Germany

This report also appears in Proc. TOOLS Pacific '97, Melbourne, Australia, November 1997.

1 Introduction

Java [Gosling 96] is being heralded as the ultimate language for internet–based distributed programming. But when it comes to actually writing distributed applications, a serious flaw in the basic design of the language becomes apparent: Java's type system, together with its mechanism of parameter passing, leads to problems with remote method invocation.

The current version of Java does not allow for a distributed implementation where remote method invocation would have the same semantics as its local counterpart. The problem is also not adequately addressed in RMI [Sun 97], Sun's proprietary technology for distributed Java programming. As a result, semantic deviations from the intended program logic can occur in distributed programs written in Java/RMI. It is therefore questionable whether Java in its current form is a suitable platform for distributed programming.

The problem is explained in section 2, which is followed by a discussion of RMI in section 3. Section 4 explores the combination of Java and CORBA [OMG 95]. Our observations suggest that revising the language itself may be the only satisfactory solution. Several options for this are discussed in section 5, and our preferred option is presented.

2 Remote parameters in Java

Java's method invocation mechanism is not suitable for distributed programming based on remote invocation. To show this, we will first sketch the syntax and semantics of purely local invocations, focussing on parameter passing, and then look at the consequences of extending this invocation semantics to the distributed case.

2.1 Java parameter passing

An ordinary Java method is invoked using the familiar notation

```
objectRef.methodName(arg1,...,argn);
```

Java, like C, has only one parameter passing mode: pass–by–value. For *primitive types* such as integer, character, etc. this means that the method receives a copy of the actual value. If any changes are made to such a copy during the execution of the method, these changes are not visible in the calling context.

If an actual parameter has a *reference type*, a reference to an object is copied, causing the object to be shared among caller and callee. Any changes the method makes to the object are visible in the calling context. In Java, all instances of classes have reference types, including strings which are instances of the library class `String` and arrays (for which no library class exists). Java provides no parameter passing mode that would dereference a parameter and deliver a copy of the object (as, e.g., Eiffel does, using *expanded* types [Meyer 97]). We say that Java "passes objects by reference, not by value".

It is important to note that since the only way of constructing user–defined types is by writing new classes, *all* user–defined types are reference types. Thus, there is no way of passing an object of a user–defined class by value.

2.2 Parameters in distributed programs

In a distributed program, object invocation may cross machine boundaries. If the caller and the callee reside on different machines, invocation must be implemented as *remote method invocation* which is the object–oriented analog to remote procedure call (RPC). Instead of passing the arguments in processor registers or on the stack, the caller must pack them into a network message and the callee must unpack them. This is called marshalling (or serialization) and unmarshalling of the parameters.

A problem arises when *references* are involved in passing parameters remotely: a reference is usually represented as a virtual address in the caller's address space; it makes no sense to dereference it in a different address space. So the reference has to be passed in a network–wide representation which can in turn be used for remotely invoking the object it refers to.

How does this apply to Java? The first observation is that there is no obstacle to implementing remote invocation for programmer–defined classes. Reference–type parameters can be passed in a network–wide representation; embedded references can be taken care of in the same manner.

A closer view, however, reveals that the resulting programming platform is of little use. Generally speaking, it is alright to distribute *services* in this way, i.e., passing references to objects that are invoked infrequently so that the increased latency of remote invocations is negligible. But what about distributing *data*, i.e. passing *arrays* and *records* (= objects without methods)? If an array of, say, 100 000 numbers is passed to a remote method for processing, the receiver just gets the reference. Processing the array may then require 100 000 remote accesses; given the latency of each individual access, this approach is plainly prohibitive (not to mention the problem of compiling array indexing into remote access). The same problem is encountered with records where — while not as drastic — it is still annoying. With *strings*, fortunately, the problem disappears. Strings are immutable objects in Java, so passing a string by reference can be *implemented* as pass–by–value because the semantics is the same.

3 RMI

Sun's Remote Method Invocation (RMI) system aims at providing mechanisms for "seamless remote invocation on objects in different virtual machines" [Sun 97] and tries to "integrate the distributed object model into the Java language in a natural way while retaining most of the Java language's object semantics". It turns out that the keyword is "most".

3.1 RMI tries...

RMI does preserve the regular Java invocation syntax, but parameters of remote operations are treated different from those of local operations. Remote passing is done as follows: If an actual parameter has a primitive type it is passed by value. If an actual parameter has a reference type and its class implements the `Remote` interface, it is passed by reference (the local reference being replaced with a network reference). If the class does not implement `Remote` but rather implements the `Serializable` interface, the object is passed by value (using serialization). If the class implements neither `Remote` nor `Serializable` an exception is raised. Arrays are serializable by default.

While this seems to solve the main problem by not promoting arrays to full remote objects and allowing for records to be passed by value rather than by reference, this approach has its own pitfalls. Consider the following example: Let `RemoteStack` denote a class for remotely invokable stacks of integers (implementing `Remote`). A stack stores its values in an array `int[] cells`. In addition to the usual stack operations such as `push`, `pop`, `top` it exports an operation

```
public int[] dump() {
       return cells;
}
```

delivering the values currently stored in the stack. There are several stacks in our distributed system, registered in a `StackDictionary` object. Our demonstration program plays with two stacks listed in the dictionary:

```
public class RMItest {

  static RemoteStack s1 = null;
  static RemoteStack s2 = null;
  static StackDictionary stacks =
            new StackDictionary();

  public static void main(String[] args)
  {
    ...
    // stacks are created at
    // different sites and
    // registered with dictionary

    s1 = stacks.get("that");
    s2 = stacks.get("this");
    s1.push(3);
    s2.push(3);
    int[] there = s1.dump();
    int[] here = s2.dump();
    there[0] = 100;
    here[0] = 100;

    System.out.print("there: "+s1.top());
    System.out.print("here:  "+s2.top());
    ...
  }
}
```

3.2 ...but fails

As `dump()` delivers an array and arrays are `Serializable` by default, the result of a remote invocation will be a copy of the stack's array. So the output of the program should be `there: 3 here: 3`

— but be also prepared to see

`there: 3 here: 100 !`

Why is this? It is possible that one of the stacks, `s2` in this case, happens to be local to the caller, which causes the standard parameter passing semantics to take effect (yes!). So a reference to the array will be delivered, causing the assignment `here[0]=100` to effectively modify the stack.

The example shows that not only can two seemingly alike objects have different behaviour but, worse, there is no way to tell them apart. This is because it is impossible to find out whether an object with a `Remote` interface is indeed remote — or actually local to the caller.

If we — inadvertently and not knowing — use a `Remote` object locally, as in the example, the operation semantics may be different from what was expected. Note that we are not referring to a different "exceptional" semantics in the face of network latency or failure, but to the successful execution of an operation.

Also note that there is indeed no way of telling remote from local objects, even if we knew that taking one for the other might cause problems: both objects have the same type and we have no way of tracking down every single object reference in a system. RMI turns out to be dangerously confusing: it gives the programmer the false impression that, although `Remote` and `Serializable` parameters are treated differently, at least all objects of a certain class, whether implementing `Remote` or not, behave alike. As we have seen, this may not be the case.

3.3 Fixing RMI?

The problem illustrated above arises because with RMI, `Remote` objects can in fact be accessed locally as well, and in this case they exhibit a subtly different behaviour.

One approach to correct this is to draw the distinction between local objects and remote objects in a more consequent fashion. For example, we might disallow local access to `Remote` objects completely. Implementing `Remote` would then mean that objects of the class can *only* be called from a remote machine. This would clearly eliminate the unpleasant behavior of the example — but seems very restrictive.

As an alternative, we could force every call to a `Remote` object, whether it is a local call or a true remote call, to have the same semantics — that of the remote case. Parameters that would have to be serialized for a remote call must also be passed by value if the call is actually local. Thus we should always get the same semantics when calling a particular object, although, just as before, we have to deal with *two* sets of semantics in general: a "local semantics" for local objects and a "remote semantics" for objects that are (even potentially) remote. If we want to make an existing class remotely accessible, we may have to change both the method implementations and the calls to those methods from other classes. If we forget any of these changes, there is no way the compiler could warn us.

Moreover, the two kinds of method invocations still look exactly the same in the code. So maybe we should even change the syntax to make it apparent that a (potentially) remote call is something different, for example:

```
object.method (arg1, ...);
```
for local semantics

```
object<-method(arg1, ...);
```
for remote semantics

Using runtime checks, the compiler would ensure that a remotely accessible object is always called using the `<-` notation and that a local object is always called using the `.` notation. Thus, the programmer would really have to think about each call, especially when converting existing code — something that seems quite consistent with the overall design of RMI.

3.4 Beyond RMI

RMI embodies a fundamental assumption about how distributed computing should be done. The chief designers of RMI take the view [Waldo 94] that remote objects are intrinsically different from local objects and that this difference must not be hidden by the distribution technology in use. In other words, they do not subscribe to the idea of *distribution transparency*, which has been favored by other researchers and which aims at just the opposite: to make the distinction between local objects and remote objects *invisible* for the programmer, so that distribution issues are hidden behind an abstraction boundary.

Distribution transparency has many facets, the most important of which is *access transparency*. We speak of access transparency when the syntax and semantics of object invocation are identical for both local and remote objects. Ideally, "remote method invocation" should just refer to a distributed implementation of otherwise unchanged (with respect to syntax and semantics) method invocation. There are numerous programming languages which achieve this ideal to a high degree, ranging from more experimental ones [Bal 92, Cardelli 95] to mainstream languages like Ada 95 [Barnes 97].

This is not to say that access transparency is an absolute value. There are areas where non–transparent technologies like RMI might be appropriate (see also [Kiczales 96, Lea 97]). RMI in its *current* form, however, is problematic for two reasons. First, there is a confusion about the concept of *remoteness*: RMI regards it as a *property* of an object to be "remote", but in fact remoteness is a *relation* between objects, as in "A is remote from B" vs. "A is local to B". Second, RMI does not, in a sense, live up to its own standards: it sets out to be deliberately non–transparent, but "pseudo–transparency" lurks in there, manifest in the strange phenomenon that the syntax of local and remote invocation is still the same — but not the semantics.

There have been efforts to hide the RMI technicalities behind a smoother facade, in order to achieve more transparency (see, e.g., [Philippsen 97]). This can, of course, work only to a limited degree, as the semantics will be invariably those of RMI.

4 CORBA and Java

If RMI is not an appropriate model for a distribution–enabled Java, what is the alternative? An obvious candidate for a distributed object model is CORBA, which we will briefly describe in this section. We will also evaluate the possibility of integrating CORBA with Java.

4.1 The CORBA object model

The OMG defines an object model for its *Object Management Architecture* (OMA) in CORBA [OMG 95] through the definition of the CORBA interface definition language (IDL). This object model has been designed in order to allow access–transparent distribution and accomodation of such diverse programming languages as C, C++, Cobol, Smalltalk, Java and others through language mappings.

CORBA defines the following rules for passing data to and from operations: all *base type* values (integers, characters, ...) are passed by value, and so are arrays and records (`structs`). All *object type* values are passed by reference. (CORBA object types are defined by specifying their interface).

CORBA object references are opaque; they do not betray the locations of objects. The behavior of an object is independent of whether it is invoked locally or remotely. CORBA defines three parameter passing modes: `in`, `out` and `inout`. `in` is pass–by–value (as described for Java above), `out` is pass–by–result and `inout` is pass–by–value–result.

Obviously, the problems with the Java object model described above do not arise in CORBA as arrays and records are passed by value. The CORBA object model in itself would allow perfect access transparency. In its present form, however, CORBA lacks another important concept — passing objects by value. The OMG has acknowledged the need to pass objects by value by issuing a corresponding request for proposals [OMG 96] which recognizes that passing an object by reference "is inappropriate for some situations, such as when an object encapsulates a data structure. In these situations, passing the object by value provides for greater efficiency."

4.2 Integrating Java with CORBA

If we integrate the CORBA object model with a "host" language according to an IDL language mapping, one of its premier values — access transparency — is compromised if this host language is not simply an extension of IDL. Unfortunately, even in the case of Java, which is conceptually very close to IDL, a number of object model mismatches arise [Brose 97].

The first of these is that Java allows the overloading of operation names while CORBA does not. Secondly, the distinction between Java's reference type strings and arrays versus CORBA's base type strings and arrays cannot be concealed: the difference between a null reference and an empty value, which can be expressed in Java, cannot be expressed in CORBA.

The most important point is, again, the difference in the respective parameter passing modes. Obviously, we cannot have both CORBA and Java parameter passing semantics as the two are mutually exclusive, so there remains a clear difference between calling objects in the CORBA and the Java object models. Hence, an access–transparent integration of CORBA with the Java language is not possible.

5 Revising Java?

We have seen that CORBA provides a suitable object model for access transparency while Java does not. We will therefore examine how Java might be modified with respect to parameter passing. We will discuss a number of options for parameter passing techniques and how the concept of passing objects by value could be realized.

5.1 Passing by reference considered harmful

The suitability of the local parameter passing modes for remote invocation has been questioned since the introduction of Remote Procedure Call. The reason is efficiency — an important consideration in a distributed system where communication can become a serious bottleneck. But then, efficiency has always been scrutinized for local parameter passing as well. Programmers (and language designers) have often taken to using by–reference where by–value would have been adequate, just for efficiency reasons. This misuse, however, works only if the parameter is not modified inadvertently, either by the receiver or through aliasing. The language should give the programmer the choice to pass objects either by value or by reference.

A conceptually attractive technique for implementing pass–by–value is *copy–on–write*: just a reference is passed, and a copy will be generated on demand, i.e., if and when the object is modified. Unfortunately, efficient implementation of this techique requires virtual memory support — unless writing is excluded a priori, i.e., the language treats value parameters as read–only and prevents the actual parameter from being modified through aliasing.

Remember that pass–by–reference is a typical device for "low–level" imperative programming (as oppposed to, e.g., functional programming) and should be shunned wherever possible. We should pass an object by reference only if the very job of the receiver is to modify the object, or to sense changes applied to the object by concurrent activities, or to serve as a container for object references. For a counterexample, consider array objects. An array is usually part of the data representation of a larger object. If it is passed as a parameter of an object invocation, it is usually passed by value. Passing by reference is confined to local auxiliary procedures of the object. This reasoning is all the more valid for small, record–like objects and for variables of primitive types (which, in Java, are passed by value anyway).

5.2 Distributed implementation

What does this imply for remote invocation? To begin with, the efficiency properties of pass–by–value and pass–by–reference are inverted. Thus, restrictions on pass-by-reference as suggested above come in handy. Let us make sure that no trouble spots remain:

1. The object in question is a typical *server* object — heavy–weight and/or immobile and/or infrequently invoked. Pass–by–reference is alright here. If the object is mobile and small, a migrational variant such as *pass–by–visit/pass–by–move* [Black 87, Achauer 93] may exhibit better performance, depending on the access pattern.

2. For *arrays*, pass–by–reference must not be allowed in remote invocations. If this restriction is not obeyed we would get an error message (e.g. from the stub compiler) when generating the distributed version. As we have seen, however, the restriction is not a serious one and barely compromises access transparency.

3. For all *other* objects, e.g. record–like data objects, passing by reference poses no conceptual problems. It may not be efficient, but, as argued above, will occur only rarely. This would even justify not supporting remote access to public attributes (again producing error messages).

In order to prepare the ground for designing an appropriate modification to Java we have to have a closer look at pass–by–value for Java objects.

5.3 Passing objects by value

Determining the parameter passing mode

One of the questions connected with passing objects by value is how the parameter passing mode for arguments of a particular operation is determined. The approach taken by RMI is to introduce new interfaces like `Remote` and `Serializable` and select the parameter passing mechanism for a particular argument according to this argument's type: if it implements `Serializable`, it is passed by value. A similar approach is taken in [Netscape 97], one of the four submissions to the OMG's object–by–value RFP [OMG 96]. Here, the notion of *stateful interfaces* is introduced, i.e. interface types that define an explicit representation for the state of an instance. Instances of stateful interfaces are always passed by value.

However, this approach — even if it is applied consistently — is inappropriate. On the one hand, it is not flexible enough as we cannot pass an object `x` by reference to an operation `f()` and by value to another operation `g()`, which might be desirable, e.g., if `f()` is designed to make a single update to a data structure embedded in `x` and `g()` needs to access this same data structure frequently, e.g. by traversing it 1000 times without altering it.

On the other hand, it is problematic that an operation cannot rely on the passing semantics used for its arguments as these may not be known at compile time. A formal parameter type `T` might have two subtypes `T1` and `T2`, one of which implements `Serializable` while the other does not. The actual argument supplied to the method invocation might be of any of the types `T`, `T1` and `T2` as Java allows polymorphism. (This is not a problem in RMI because RMI, as shown above, uses serialization for remote calls only. The RMI stub generator complains if a remote interface extends another, non–remote interface. Thus, it keeps the inheritance hierarchies for remote and non–remote types separate so that the above problem caused by polymorphism cannot arise. For a more general approach, though, this remains a problem.)

The underlying misconception of the whole approach is that the mechanism for passing a parameter to an operation is selected on the basis of the type of the actual argument, so the semantics of the passing mode is regarded a property of this *type* while in fact it is part of the *operation semantics*. Consequently, it should be expressed in the declaration of the operation's formal parameters and its return type, as already argued for the local case in 5.1.

A new keyword for specifying pass–by–value

We propose the introduction of a new keyword `copy` which specifies that a parameter is to be passed by value. Consider the following method declaration:

```
public copy int[] aMethod (copy A a, B b, int i)
```

The parameter passing modes for `myMethod` are as follows: Both the return value of reference type `int[]` and the formal parameter `a` of reference type `A`, are declared using `copy`; this means that the corresponding objects are passed by value. Using `copy` does not change the fact that both the formal and the actual parameter are references. The only effect is that, e.g., `a` refers to a newly created copy of the actual parameter. Of the remaining parameters, `b` is passed by reference and `i` is passed by value.

It is important to remember that, according to 5.2, omitting the first `copy` would cause an error message to be produced during the generation of the distributed version. Omitting the second `copy` would produce an error message only if `A` had public attributes and the system would not support remote attribute access.

The semantics of `copy` is similar to the semantics of `expanded` in the Eiffel language [Meyer 97]. In Eiffel, `expanded` is used to declare variable names that denote objects, not references to objects. It can also be used to specify pass–by–value semantics for arguments and return values of reference types. However, if a parameter is passed `expanded`, the receiver will get a *copy* rather than a *reference* to a copy.

Serializing object state

To support object copying in a homogenous environment, a single implicit serialization/deserialization protocol realized by the distribution platform could be used. RMI, e.g., provides such a mechanism. This protocol would specify what parts of the object are to be copied in which order. Such an *implicit* protocol would not, however, suffice in heterogeneous environments such as CORBA where interoperability between implementations and platforms is a major concern. In order to realize object–by–value semantics in such a setting, the protocol used for serialization/deserialization of objects needs to be explicitly defined. Alternatively, a negotiation mechanism for selecting a serialization protocol could be specified. In the following we will assume some reasonable default protocol.

Copies made as a consequence of passing an object by value are "shallow copies" by default, i.e. objects contained as references within the object are not copied; instead those references are automatically replaced by network references. If this behavior is not appropriate for a particular situation, the class of the object that is passed must inherit from a special interface and the programmer needs to provide the specific copying mechanism.

We assume that an implementation for the appropriate object type (a Java class) is available in the receiving context or can be made available by downloading Java code if necessary. But which is the appropriate type? In order to support polymorphism properly, the receiving context needs to be given an object of the most derived type of the actual argument supplied in the invocation.

6 Conclusion

We have shown that Java, although explicitly designed with the network in mind, exhibits weaknesses in an important network–related issue and defies access transparency more fiercely than traditional languages. A number of approaches to solving the parameter passing problem have been presented, emphasizing the need for a solution that avoids distribution–related featurism.

A modest modification of Java has been proposed: a new keyword `copy` for specifying pass–by–value semantics for reference type parameters. It has been argued that carefully choosing the semantically appropriate parameter modes makes a program fit for both centralized and distributed execution. Not surprisingly, the resulting mechanisms are in line with CORBA's model of parameter passing. An interesting perspective would be to evaluate Java as a starting point for designing a CORBA *implementation* language.

References

[Achauer 93] Bruno Achauer:
The DOWL distributed object–oriented language, *Communications of the ACM*, **36**(9), September 1993, 48–55.

[Bal 92] Henri Bal, Frans Kaashoek, Andrew Tanenbaum: Orca: a language for parallel programming of distributed systems, *IEEE Trans. on Software Engineering* **18**(3), 1992, 190–205.

[Barnes 97] John Barnes: *Ada 95 Rationale: The Language, The Standard Libraries*, Springer, 1997.
`http://www.adahome.com/Resources/refs/rat95.html`

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, Larry Carter: Distribution and Abstract Types in Emerald, *IEEE Transactions on Software Engineering*, **13**(1), January 1987, 65–76.

[Brose 97] Gerald Brose: JacORB — Design and implementation of a Java ORB, *Proc. Int. Conf. on Distributed Applications and Interoperable Systems DAIS'97*, Cottbus, Germany, September 1997, Chapman & Hall, 143–154 .
`http://www.inf.fu-berlin.de/~brose/jacorb/`

[Cardelli 95] Luca Cardelli: A language with distributed scope, *Computing Systems* **8**(1), January 1995, 27–59.
`http://www.research.digital.com/SRC/personal/Luca_Cardelli/Obliq/Obliq.html`

[Gosling 96] James Gosling, Bill Joy, Guy Steele: *The Java Language Specification*, Addison–Wesley 1996.
`http://java.sun.com/`

[Kiczales 96] Gregor Kiczales: Beyond the black box: open implementation, *IEEE Software* **13**(1), January 1996, 8–11.

[Lea 97] Doug Lea: Design for open systems in Java. *Proc. 2. Int. Conf. on Coordination Languages and Models*, LNCS 1282, Springer 1997, 32–45.

[Meyer 97] Bertrand Meyer: *Object–Oriented Software Construction*, 2nd ed., Prentice–Hall 1997.

[Netscape 97] Netscape, Novell, Visigenic: *Objects By Value, Joint Initial Submission*, OMG TC document orbos/97–04–02.
`ftp://ftp.omg.org/pub/docs/orbos/97-04-02.ps`

[OMG 95] OMG: *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1995.
`http://www.omg.org/`

[OMG 96] OMG: *Objects–by–Value RFP*, OMG document orbos/96–06–14, June 1996.
`ftp://ftp.omg.org/pub/docs/orbos/96-06-14.ps`

[Philippsen 97] Michael Philippsen, Matthias Zenger: JavaParty — transparent remote objects in Java, *Proc. ACM PPoPP Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997.
`http://wwwipd.ira.uka.de/~phlipp/party.gs.gz`

[Sun 97] Sun Microsystems: *Java Remote Method Invocation Specification*, Revision 1.4, 10 February 1997, Sun Microsystems 1997.
`http://java.sun.com/products/jdk/ 1.1/docs/guide/rmi/spec/ rmiTOC.doc.html`

[Waldo 94] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall: A note on distributed computing, Sun Microsystems Technical Report TR–94–29, November 1994.
`http://www.sunlabs.com/technical-reports/1994/abstract-29.html`