JacORB: Implementation and Design of a Java–ORB

Gerald Brose
Institut für Informatik,
Freie Universität Berlin
Takustraße 9,
D-14195 Berlin, Germany,
brose@inf.fu-berlin.de

Abstract

We describe a number of experiences we have made designing and implementing JacORB, an Object Request Broker (ORB), in Java. These are mainly consequences of choosing Java both as the implementation language for an ORB and as the target language for which this platform is designed. We focus on the degree of transparency possible in a Java language mapping for IDL, on multi–threading in the ORB core and generating CORBA type code information.

Keywords

Java, CORBA, transparency, multi–threading, type codes.

1 INTRODUCTION

JacORB is a partial implementation of the OMG CORBA standard (OMG 1995). It was developed at Freie Universität Berlin as a freely distributable* ORB with the aim of making CORBA technology easily accessible and providing an implementation of an ORB which could be used as a kind of textbook example for teaching basic ORB technology and principles. Thus, making the JacORB implementation intelligible and keeping the design as simple as possible was of major importance. In general, when design decisions had to be made with tradeoffs between performance and simplicity, we tended to disregard performance. Having access to an ORB at the source code level has already proved valuable in an advanced programming course on distributed systems.

In order to make a CORBA platform widely accessible, we chose the Java virtual machine (Gosling *et al*. 1996, Lindholm *et al*. 1997) as our target architecture and restricted the implementation language to *pure* Java, i.e. we do not use any "native code". Consequently, JacORB is available on all major hardware and operating sys-

*JacORB can be obtained from `http://www.inf.fu-berlin.de/~brose/jacorb/`.

tem platforms and ensures an equally wide portability of applications written with JacORB.

In this paper, we first give an overview of JacORB and then describe a number of experiences we have made designing and implementing an Object Request Broker in and for the Java programming language.

Developing an ORB implementation as a Java programming platform, one important question that arises is how closely Java and CORBA can be integrated on the language level. The similiarity between the CORBA and Java object models allows for a rather straightforward language mapping and, as a consequence, a high degree of transparency for developers when compared to ordinary Java programming. This is the subject of section 3.

Choosing Java as the implementation language for an ORB had a number of interesting implications beyond portability. With the language–integrated thread model and the reflection features of Java JDK 1.1, handling concurrency in the ORB kernel and the generation of CORBA `TypeCodes` were much simpler than we had expected. These issues are presented in sections 4 and 5.

2 JACORB ARCHITECTURE

JacORB comprises an IDL–compiler, a stub generator, a runtime system and two prototypical object service implementations — *Naming* and *Events* — as specified in (OMG 1994). Stubs and skeletons have to be statically compiled from IDL specifications with the IDL compiler and a separate stub generator. These proxy classes are dynamically linked with client and server programs. The ORB core is library–based, i.e. all of the CORBA functionality realized by JacORB resides in the libraries. In order to allow multi–threading in both clients and servers, the JacORB libraries are threaded. The JacORB implementation currently consists of approximately 150 central classes with 13 500 lines of code.

In its present form, JacORB requires no instance of a separate runtime component on network nodes running JacORB applications. All that is needed is an instance of the JacORB name server, which is used to locate objects, and an HTTP server in the same domain, which is used to enter a name space by locating the name server via a URL.

In the present release 0.6, there is no *Dynamic Invocation Interface* (DII) and no *Dynamic Skeleton Interface* (DSI), no *Interface* and *Implementation Repository* and no separate *Object Adapter* component. The implication of not having a "real" *Object Adapter* at the moment is that there is no automatic activation of servers for object implementations and no mechanism for selecting different activation modes, i.e. servers have to be up and running at the time a request is made. Other adapter functionality like dispatching requests is carried out by the communication subsystem while method dispatch is performed in the skeletons.

Interoperability with other CORBA implementations is achieved through the internal use of *Interoperable Object References* (IOR) and the *Internet Inter–ORB Protocol* (IIOP). This also implies the use of the *Canonical Data Representation* (CDR) as transfer syntax. IIOP is mandatory for CORBA 2.0 compliance and implemented in the JacORB transport layer. Interoperability has been successfully tested with objects running on other ORBs such as Orbix, OrbixWeb, NEO and OmniBroker. The performance of the transport layer is, even without any optimization, within the range of commercial Java products. On the average, remote method invocations are between 2–3 times slower than with OrbixWeb (IONA 1996b) and significantly faster (up to factor 9 for invocations with large amounts of data) than with Sun's RMI (Sun 1997) system.

The IDL–compiler realizes the language mapping described below and generates a number of Java classes during this process, but it does not generate stubs and skeletons. This is left to a separate tool, the generator. The generator takes Java `.class`–files as input, so the source code output from the IDL–compiler has to be compiled with a Java compiler before it can be fed into the generator. As a simple example, consider the following IDL specification:

```
module example {
   interface server {
      void writeMessage(in string a1);
   };
};
```

The IDL–compiler will generate a number of Java classes from this specification. The most important of these is the Java interface `server`:

```
package example;
public interface server extends CORBA.CORBject {
   void writeMessage(java.lang.String a1);
}
```

After compiling the Java interface, the JacORB generator would be used to generate a stub and a skeleton file for the interface. This separation between mapping IDL to Java and generating stubs and skeletons is useful for a number of reasons. First, it separates concerns and makes the individual tools more managable. This is important as the IDL–compiler has grown to become the most complex component of JacORB and accounts for approximately 50% of the whole code. Keeping the IDL–compiler as simple as possible was also a major motivation for looking at alternatives to using it for generating `TypeCode`–information, as will be explained in a later section. Second, this separation allows the generator to be used for distributed Java programming without IDL specifications (which is the purpose for which JacORB was first designed). The generator can take advantage of the compact and well documented canonical Java class–file format and need not perform any parsing of source code on its own.

JacORB provides developers with an easy–to–use programming interface which is sufficient in most cases where a full CORBA compliant API is not needed. The following code is the simplest possible example for a client program accessing a remote object through its name. The static method `locateObject()` in generated class `serverStub` uses the JacORB name service in its implementation in order to locate the object and return a reference to it. It is a merely convenient shortcut for using the standard COSS interface to the naming service. For simplicity, the name service uses a separately configured default naming context to resolve the object name in this case, but it could have been directed to query other name spaces as well by supplying more parameters to `locateObject()`.

```
package example;
public class Client{
   public static void main(String[] args) {
      server s = serverStub.locateObject("server");
      s.writeMessage(args[0]));
   }
}
```

The result of the `locateObject()`–invocation is an opaque object reference to the object implementation realized by a proxy object — the stub. This stub is already connected to the object implementation and ready to receive invocations. For an object to be locatable via a name, it needs to be registered with the name server. As an example, consider the following piece of code which shows a server program holding the object implementation `serverImpl`. A skeleton for this object is created and then registered. Again, the statically generated class provides a simple shortcut to explicitly using the name service. The object implementation itself, i.e. the class `serverImpl` in the example, is a regular Java class without any JacORB–specific code.

```
package example;
public class remoteServer {
   public static void main(String[] args) {
      serverImpl s = new serverImpl();
      serverSkeleton st = new serverSkeleton(s);
      try {
         st.registerService("server");
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Object references can be passed around as arguments for operations and can be obtained as results of operations. JacORB hides all the complexity and always provides ready–to–use stubs bound to the object implementations, thus insulating programmers from distribution details. Local invocations do not involve any stubs, and there will only be one stub object in an address space for any one remote object. JacORB takes care that no superfluous stubs are created or used.

3 TRANSPARENCY

JacORB aims at *transparently* integrating Java's non–distributed object model with the more general CORBA object model. In order to make distribution totally transparent to programmers, remote object invocations need to be made syntactically and semantically indistinguishable from local invocations. If introducing the CORBA object model by means of an IDL–language mapping can be accomplished while preserving the syntax and semantics of object invocations in the target language, we get perfect transparency at the language level.

It should be noted that this is a different approach than the one taken in RMI. Remote method invocations in RMI have different semantics than non–remote invocations with respect to parameter passing: Objects which implement the `Remote` interface are passed by reference in remote calls whereas all other objects are passed by value, while objects are always passed by reference in local invocations. Moreover, remote invocations always potentially throw a `RemoteException`, so that they have to be embedded in `try-catch` blocks. This is not, of course, the case for local operations.

It has already been pointed out above that the Java object model is not very different from the CORBA object model. Indeed, the CORBA object model can mostly be regarded as an extension of the Java object model if we make one assumption regarding object types in Java. We require that those object types whose instances are to be distributable in Java are defined as interfaces and not as classes. This is a necessary precondition, because JacORB requires that stubs generated from interfaces are subtypes of these interfaces so that they are assignable to variables of this interface type.

CORBA concepts not present in Java have to be modelled using Java language constructs. If no mismatches between CORBA and existing Java concepts arise, transparent integration of Java programs into CORBA environments is possible. As it turns out, there are surprisingly few of these mismatches.

In the remainder of this section, we will first sketch the IDL language mapping used in JacORB and then focus on those points that touch on transparency.

3.1 The Java language mapping

The OMG issued an RFP for a Java language mapping in August 1996 (OMG 1996) to which there is a joint submission by all major ORB vendors (DEC *et al.* 1997). The final release of a corresponding specification is expected in the second quarter of 1997. Any language mapping that we might use must therefore be considered preliminary, but as the mapping is mostly straightforward, we do not expect too many changes once the OMG mapping is agreed upon. The main design rationale in our mapping is to make it natural and easy to use for Java programmers as well as easy to implement.

In general, the basic IDL data types are mapped onto basic Java types, structured IDL types are mapped onto Java classes and IDL interfaces are mapped onto Java interfaces. A few issues such as mapping IDL `typedefs`, parameter passing modes, bounded arrays and sequences, `oneway` calls and IDL name scopes require special treatment.

IDL arrays must be declared with fixed bounds whereas this is not possible in the declaration of Java arrays. We therefore map IDL arrays to Java classes containing a size attribute. Unbounded IDL sequences map to simple Java arrays. As an example, consider the following IDL interface:

```
module idltest {
   interface server {
      typedef short nums[2];
      oneway void notify(in nums n);
   };
};
```

The IDL–compiler generates two classes from this interface, one for the array declaration and one for the interface. The generated array class is the following:

```
// Automatically generated by idl2java from array type nums
package idltest;
public class servershortArray2 implements jacorb.Orb.Array {
   public short values[] = null;
   public static final int size = 2;

   public servershortArray2(){
      values = new short[size];
   }
   public servershortArray2( short _array[])
      throws jacorb.Idl.ArraySizeMismatchException
   {
      if( _array.length != size )
         throw new jacorb.Idl.ArraySizeMismatchException();
      values = _array;
   }
   public Object values(){
      return values;
   }
   // serialization code ...
}
```

The Java interface generated from the IDL interface definition looks like this:

```
// Automatically generated by idl2java from interface server
package idltest;

public interface server extends CORBA.CORBject{
   void notify(idltest.servershortArray2 n);
   boolean notify$oneway = true;
}
```

The IDL keyword `oneway`, which declares a different invocation semantics for operations, leads to one constant attribute per `oneway`–operation in the generated Java interface. This attribute is used by the runtime system in order to determine that a different semantics is to be used for invocations of `notify()`. Since operations in JacORB return to a different thread on the calling side anyway, `oneway`–semantics can be achieved by simply not synchronizing with the return of the operation as it would be done in the case of regular operations. This causes no problems as CORBA prescribes that `oneway`–operations may have neither return values nor `out`– or `inout`–parameters, nor may they raise user–defined exceptions.

Another problem illustrated in the above example is caused by different scoping constructs in IDL and Java. New scopes are introduced in IDL by modules as well as by interfaces or structs. Java, however, has only packages.* The visibility of class names within a Java package cannot be further constrained. Thus, mapping IDL identifiers to Java can potentially lead to name clashes. In order to avoid such name clashes, names are prefixed by the names of the IDL scopes they are contained in. As a result of this rule, the name of the generated array class above is `servershortArray2` and not simply `shortArray2` as it is declared local to the scope introduced by the interface `server`.

Since Java has only one parameter passing mode — `in` in IDL —, the other two IDL parameter passing modes `out` and `inout` have to be imitated by generating wrapper classes or *Holders* for every IDL type. For every paramter declared `out` and `inout` in IDL, the corresponding operation in the generated Java interface will list the respective *Holder*–class as the parameter's type.

More details on the Java language mapping employed in JacORB can be found in (Brose 1997).

3.2 Transparency restrictions

The CORBA object model is not simply an extension of the Java object model, there are also a few mismatches between the two. The most prominent restriction from the point of view of a Java developer is the fact that IDL does not allow overloading of

*This applies only to Java 1.0.2. JDK 1.1 allows nested classes.

operation names. However, programs using overloaded names can be transformed into programs without overloaded operation names by systematic renaming.

Another restriction arises from the fact that strings and arrays have object types in Java but basic types in CORBA IDL. Values of these types are passed *by-reference* in Java whereas they have to be passed *by-value* in CORBA. Apart from different semantics in the case of arrays*, the distinction between an empty string or array and a null–string or array cannot be expressed, because null references must not be used in remote invocations whereas this would be allowed in local calls.

The last point that is different in Java–programs written for CORBA environments is the fact that Java does not support unsigned integer types, so large positive values of mapped unsigned CORBA IDL types may result in negative values in Java.

None of these issues is critical as such, but they all serve to distinguish purely local Java operation invocations from remote invocations.

4 MULTI–THREADING IN THE ORB CORE

Distributed systems are generally concurrent, even if clients and servers are themselves sequential because any number of clients could access a server at the same time. Another potential source of concurrency is the ability to create threads in Java programs. Thus, concurrency can arise from multiple clients invoking operations of an object implementation simultaneously, or it can arise from multiple threads in one client. Obviously, it would be undesirable if an ORB implementation automatically serialized the execution in different threads, so ORBs have to offer some sort of multi–threading support.

While Orbix (IONA 1996a, IONA 1995) delivers a different set of libraries for multi–threaded programs than for single–threaded ones (at significantly higher prices), we regard the ability to program multi–threaded servers as generally indispensable. Moreoever, the integration of threading into the language makes the task of providing proper support for multi–threading much simpler than with other languages. JacORB is therefore designed to handle multi–threading in the ORB core, so no user intervention is necessary unless concurrency is explicitly to be suppressed. However, programmers do have to insert synchronization code into object implementations if concurrent access could potentially lead to inconsistencies because JacORB always exposes object implementations to concurrent access.

While this may be regarded as extra complexity burdening developers, providing multi–threading support by default on the other hand relieves developers from having to deal with another possible source of errors. Without a threaded ORB core, callbacks from a server to a client will lead to deadlock, so developers would have to judge whether such calls are possible and then decide which set of libraries to use. The following paragraphs elaborate on multi–threading in the JacORB core.

In order to allow concurrent activity in clients, it must be guaranteed that the runtime system does not block waiting for operation replies. Only the individual client

*Java–strings are immutable, so there is no difference between *pass–by–value* and *pass–by–reference*.

threads need to block after invoking an operation until the invocation returns. This requires the reception of reply messages to be decoupled from the sending of requests, so JacORB uses a separate thread to receive replies. Sending of requests is done within the individual client's threads of control.

While a single object reference is only ever used for accessing one object implementation and thus only needs to manage one connection, the situation is different on the server side. A skeleton may receive requests on behalf of its object implementation from many different clients. Connection management on the server side therefore uses two layers of threading. Every skeleton has one instance of class `NetServer` listening on a port for connection requests. Once such a connection is established, the `NetServer` creates a separate instance of itself running in a different thread. This thread is responsible for receiving request messages over a single connection while the first `NetServer` continues to listen for additional connection requests from other clients.

In order not to suppress interleaving of operation invocations from multi–threaded clients, the `NetServer` object responsible for a given connection needs to employ threading again — if the invocation was actually executed in the thread of the `NetServer`, other incoming requests from the same source would have to wait for its completion. The `NetServer` starts a different `Skeleton`–thread for every incoming request message ("thread–per–request" model).

To sum up, JacORB has one instance of class `NetServer` listening on a TCP port for every object implementation. Every such instance starts up another `NetServer`–thread per connection, i.e. per client address space. This thread in turn starts `Skeleton`–threads for every incoming request message.

5 TYPECODES

CORBA defines `TypeCodes` as a data structure for providing run time type information about IDL types. This information is used as one component of the generic type `any` so that users of `any` values can determine what IDL type the value actually has. `TypeCodes` are also used in the *Interface Repository*.

In order to make this kind of meta–information available at runtime, two approaches are possible: `TypeCodes` can be generated at compile time and inserted into the Interface Repository by the IDL–compiler, or the IDL–compiler could include extra information in target language constructs when mapping IDL to this language.

Generally, the first approach is inflexible and might not have produced the necessary meta–information because it was not foreseen at compile time* that it could be needed, while the second approach might result in unnecessary overhead because too much information is included which might never be used. Both approaches demand that the IDL compiler perform extra work while mapping IDL to the target language.

*Assuming, of course, that the IDL compiler does not automatically generate type codes for *every* IDL type and includes these in the Interface Repository, but rather has a compiler option to be used in this case.

Using the reflection features of Java 1.1, we found that the IDL compiler can be relieved of most of this kind of extra complexity. Moreover, the flexibility of the second approach above can be achieved with only neglegible overhead. Effectively, what is needed is a *reverse mapping* between Java and IDL which can be applied without much runtime overhead — such as invoking a compiler and generating new class files.

This is possible as the JDK 1.1 allows runtime access to all features of an object's class: methods, fields, superclasses, type name, constructors and component type (if it represents an array type, e.g.). This information can be used to construct `TypeCodes` at runtime if classes generated by the IDL compiler can be distinguished from ordinary Java classes and if all classes generated from IDL structured types can be told apart so as to be able to distinguish a class representing an IDL `struct` from a class representing an IDL `interface`. The IDL compiler need only include a little extra information like `class X implements jacorb.Orb.Struct` in class declarations so that this distinction can be made on the basis of the supertypes of a class or interface.

The generation of `TypeCodes` can thus be implemented in a relatively simple static method `getTypeCode()` in the JacORB class `TypeCode` which returns a `TypeCode` object for a given class. This method is used by the JacORB class `Any` when inserting values into an `Any` and marking its type with a `TypeCode` object. While we have not yet built an implementation of the *Interface Repository*, we expect to be able to realize it without further support from the IDL compiler, relying on the approach outlined above.

6 SUMMARY AND FUTURE WORK

We have presented JacORB as a Java implementation of the major part of CORBA and reported on a number of experiences made during the design and implementation process. These include conceptual issues when mapping IDL to Java as well as practical consequences of our choice to implement JacORB in Java, such as multi–threading support in the ORB core and runtime generation of `TypeCodes`.

JacORB supports the integration of Java and CORBA by providing a mapping from IDL specifications to Java. Another interesting option that we intend to investigate is the possibility to generate IDL specifications from existing Java code as suggested in (OMG 1997). JacORB already allows distributed Java programs to be written that use IIOP for communication. By reverse–mapping Java to IDL, existing Java applications could be made visible in CORBA environments without the need to rewrite code.

One of the main goals for future JacORB releases is to develop an object adapter and an Implementation Repository. Part of the object adapter functionality is presently realized by the skeletons, but a more flexible registration and activation of object implementations would be desirable. The introduction of a separate object adapter component will probably lead to a JacORB daemon process on every network node on which servers are to be run. We have, however, not decided whether JacORB is to support the *Basic Object Adapter* (BOA) as specified in (OMG 1995) or rather take the specification of the *Portable Object Adapter* (POA) from (BEA *et al.* 1997) into consideration. The *Interface Repository* will most likely be realized as one extra process for an ORB domain and could make use of Java's reflection features as has been pointed out in the previous section.

JacORB could be made more CORBA–compliant in a few places. The IDL language mapping for Java will probably differ from our mapping in a number of respects, and we could try to align the JacORB mapping to the official mapping. The interface of the ORB itself does not presently conform to the CORBA standard where operations such as `ORB_init(), BOA_init()` and `list_initial_services()` are specified, but this is not particularly difficult to achieve. Our main interests in future research with JacORB is to further enhance its usability and the degree of transparency possible for CORBA developers. Especially on the server side, possible tool support for generating the lower level source code for server processes appears to be an interesting perspective.

REFERENCES

BEA, DEC, Expersoft, HP, ICL, IONA, Novell, SunSoft, Telefonica I+D (1997) *ORB Portability Joint Submission, Draft 14*, OMG document orbos/97–04–04.

Brose, G. (1997) *JacORB — A Java Object Request Broker*, Technical Report B–97–2, Freie Universität Berlin, April 1997.

Brose, G. and Bokowski, B. (1997) Ein Object Request Broker für Java, *Informatik/Informatique*, Zeitschrift der schweizerischen Informatikorganisationen, **No. 3/97**, June 1997, 27–30.

DEC, Expersoft, HP, IBM, Netscape, Novell, Oracle, Sun, Visigenic, Xerox (1997) *IDL/Java Language Mapping, Joint Revised Submission*, OMG TC document orbos/97-02-01.

Gosling, J. and Joy, B., Steele, G. (1996) *The Java Language Specification*, Addison–Wesley.

IONA (1995) *Orbix 2 Programming Guide*, Release 2.0, IONA Technologies Ltd.

IONA (1996a) *The Orbix Architecture*, IONA Technologies Ltd.

IONA (1996b) *OrbixWeb Programming Guide*, IONA Technologies Ltd.

Lindholm, T. and Yellin, F. (1997) *The Java Virtual Machine Specification*, Addison–Wesley.

OMG (1994) *Common Object Services Specification, Volume I*, Revision 1.0, March 1994, OMG Document 94–1–1.

OMG (1995) *The Common Object Request Broker: Architecture and Specification*, 2.0, July 1995.

OMG (1996) *Java Language Mapping RFP*, OMG TC document orbos/96-08-01.

OMG (1996) *Java to IDL Request For Proposal*, OMG document orbos/97-03-08.

Sun (1997) *Java Remote Method Invocation Specification*, Sun Microsystems, Inc.