JacORB — A Java Object Request Broker

Gerald Brose

brose@inf.fu-berlin.de

Technical Report B 97–2

April 1997

Abstract

This document describes the architecture, design and implementation of JacORB, a free and portable object request broker written in Java. JacORB is a partial implementation of CORBA, the OMG's Common Object Request Broker Architecture. The current version as of this writing is 0.6.

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

Contents

Chapter 1

Introduction

JacORB is a free[1] implementation of OMG CORBA and was developed at Freie Universität Berlin. It has been used for developing CORBA applications as well as for teaching CORBA.

While not providing the entire funcionality of CORBA 2.0 with respect to dynamic invocations, JacORB supports most of CORBA's static invocation features and allows multi–threaded clients and servers to be written in Java[GJS96]. It offers CORBA interoperability by implementing CORBA's *Internet Inter–ORB Protocol* (IIOP) and includes an implementation of a name and an event service. There is no native code in JacORB, so it can be used directly on any platform for which a Java virtual machine implementation is available. Full source code and a number of examples are included in the JacORB distribution.

This paper is organized as follows. Section 2 gives an introduction to CORBA, the third section describes the JacORB architecture and how it is related to CORBA. Section four presents design issues and section five describes the implementation of the JacORB services naming and events. The concluding section 5 outlines future work. For a more practical introduction to using JacORB refer to [Brose97].

[1] according to the terms of the GNU public license.

Chapter 2

CORBA

This chapter gives a short introduction to the Common Object Request Broker Architecture. For more comprehensive introductions to CORBA see [OHE96, Siegel96]. We go on to sketch a few different architectures for implementing an Object Request Broker.

2.1 A CORBA introduction

CORBA [OMG95] defines the infrastructure for OMG's Object Management Architecture (OMA) by providing standards for object invocations in heterogeneous environments. CORBA defines its own object model which is manifested in the *Interface Definition Language* (IDL)

IDL is used to define the interfaces of objects. The implementation of these objects can be done in any programming language provided all concepts of CORBA IDL are mapped to that language. The OMG defines a number of language mappings for the most widely used programming languages, e.g. C, C++, Smalltalk. A Java mapping is being worked on, but there is presently no official OMG standard.

CORBA objects are connected by an *Object Request Broker* (ORB) which allows objects to communicate without regard to object location. Figure 2.1 illustrates the main components of an ORB as defined by CORBA.

CORBA environments provide two ways of invoking operations on object implementations[1]: clients can use pre–compiled stubs which are automatically generated from IDL specifications, or they can use the *Dynamic Invocation Interface* (DII) to build operation invocations at runtime.

On the server side, these different approaches are matched by statically compiled *Skeletons* or by the *Dynamic Skeleton Interface*. While the client side functionality is actually not very complicated and can be provided with a high degree of transparency,

[1] The term *server* is avoided here because a server in CORBA parlance refers to an operating system process in which active object implementations reside. Services are provided by these object implementations.

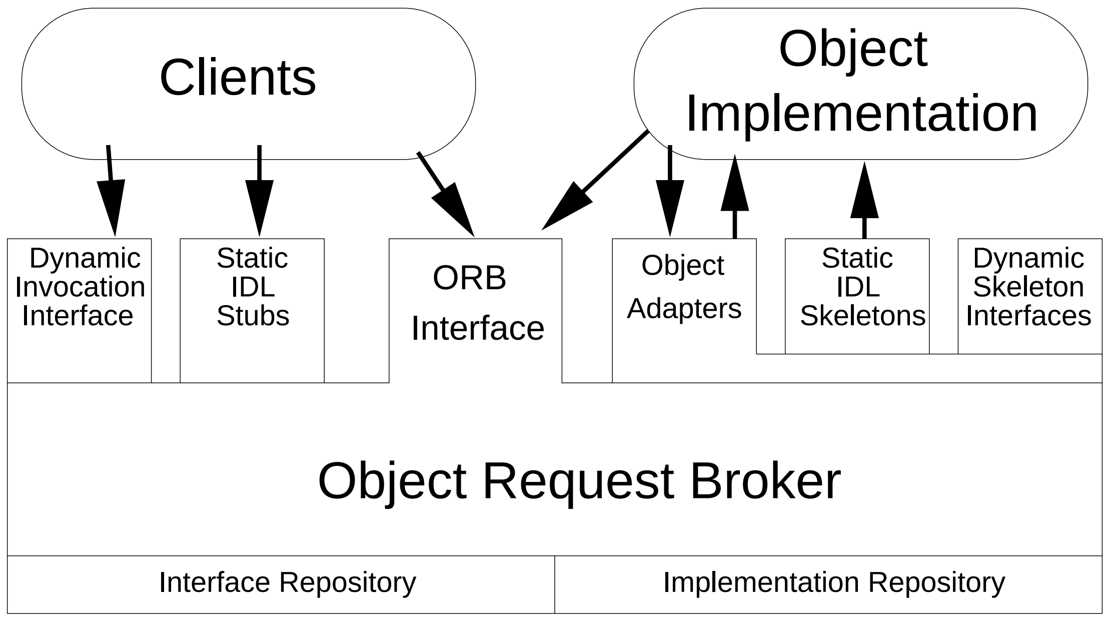

Figure 2.1: ORB architecture

the same does not hold for the server side. Since there can be no single mechanism for activating and binding to object implementations with all conceivable requirements for their runtime environment, these issues are dealt with by a separate component called *Object Adapter*. *Object Adapters* can be defined according to the special requirements implementations might have, e.g. persistent database objects might require their own adapter. The only object adapter defined in CORBA is called the *Basic Object Adapter* (BOA).

The *Interface Repository* is basically a type management component that can be used to define, alter and query type information statically and dynamically. The *Implementation Repository* is used by the Object Adapter to register object implementations.

While it is not necessary to prescribe a standard for ORB–internal communication, the scenario is different for communication between objects managed by different ORBs. These ORBs have to interoperate in such cases, so CORBA specifies the *General Inter–ORB Protocol* (GIOP) that ORBs have to implement. GIOP defines a number of message formats and a *Canonical Data Representation* (CDR) for transferring data. GIOP may be realized using any transport protocol, but a TCP/IP implementation — the *Internet Inter–ORB protocol* (IIOP) — is obligatory for any CORBA compliant ORB.

2.2 Object Request Broker Architectures

The CORBA specification does not prescribe any particular implementation of an ORB, nor does it stipulate the architecture of an ORB implementation. A number of different architectures are conceivable and are actually mentioned in [OMG95]. We briefly describe the three basic choices for ORB architectures.

Server–based ORB

The first possibility is for an ORB to be realized as a normal program which functions as a server to both client and server objects. Communication with the ORB could be done via any IPC mechanisms available. The advantage of this approach lies in the centralization of the ORB management: if any policies (e.g. security) are to be applied to all object invocations in a domain, this is the easiest way to enforce them.

Library–based ORB

While the ORB–server approach leads to a highly centralized ORB, a library–based approach effectively decentralizes ORB functionality. Here, the ORB is not a separate, identifiable run–time component but rather resides in libraries through which its functionality is made available to client and server programs.[2] One example of such an ORB implementaion is IONA's Orbix[IONA96].

Different instances of applications in the same domain could thus be made to run with very different configurations of ORB functionality. For instance, one application could be linked with libraries providing a distributed and multi–threaded environment plus security features such as encryption on the transport level, while another application could be configured to run both client and server objects in a single address space, avoiding run time marshalling of operation arguments and results and the need for encrypting messages altogether while perhaps relying on dynamic type management features of the Interface Repository which were not included in the first application.

System–based ORB

A third possible architecture for implementing an ORB could be to provide the ORB functionality as part of the underlying operating system, as in Sun's NEO[Sun97]. The advantages this approach offers are possibly enhanced security and performance as the operating system features can be used more directly in the design and implementation of the ORB. The OS could provide for authentication and could be used to avoid marshalling if client and server object are located on machines of the same hardware architecture.

[2] A daemon process, however, will in most cases still be needed on machines running server processes.

Chapter 3

A JacORB Overview

Before going into the details of the JacORB architecture and design, we state the principles and goals underlying this design.

3.1 JacORB design rationale

The two main JacORB design goals are simplicity, portability and ease of use. In general, when design decisions have to be made where there are tradeoffs between performance and one of these goals, we tend to disregard performance. Our aim was to make CORBA technology easily accessible to everyone and to provide an example implementation of an ORB which could be used as a kind of textbook example for teaching basic ORB technology and principles. Making the JacORB implementation intelligible is a main motivation for keeping the design as simple as possible.

Making a CORBA platform widely accessible touches on portability in our case, since chosing the Java virtual machine as our target architecture makes JacORB immediately accessible on all major hardware and operating system platforms and ensures an equally wide portability of applications written with JacORB. To achieve this, we needed to restrict the implementation language to *pure* Java, i.e. we do not use any "native code".

Simplicity and ease of use are equally high ranking concerns in the JacORB design. Ease of use for developers is achieved by providing a high level of distribution transparency to the programmer through the JacORB API and the IDL–to—Java language mapping. For details please see [BB97] and the section on the Java language mapping below.

3.2 JacORB architecture

A library–based ORB architecture was the obvious choice in our case. A system–based architecture was impractical as we have no means of building an ORB layer into existing operating systems. Furthermore, this approach would not have been conducive to an easy understanding of the system as a whole. A server–based approach would have been possible but appeared too inflexible and seemed to incur more communication overhead than necessary.

The JacORB architecture, thus, is library–based: all of the CORBA functionality realized by JacORB is provided by stubs and skeletons which, by inheritance, use the JacORB class library. Stubs and skeletons have to be statically compiled from IDL specifications with the IDL compiler and a separate stub generator. These proxy objects are dynamically linked with client and server programs. In its present form, JacORB requires no separate runtime component on network nodes running JacORB applications. All that is needed is one HTTP server somewhere in the domain and one instance of the JacORB name server.[1]

In the present release 0.6, there is no DII and no DSI, no *Interface* and *Implementation Repository* and no separate *Object Adapter* component.[2] The implication of not having a "real" BOA 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, and the only activation mode is "unshared server" (i.e. one object per server). Interoperability with other CORBA implementations is achieved through the internal use of the *Internet Inter–ORB Protocol* (IIOP), which is mandatory for CORBA 2.0 compliance. JacORB's interoperability has been successfully tested with objects running on Orbix, NEO and OmniBroker.

3.2.1 Stubs and Skeletons

There are two approaches to generating stubs and skeletons in JacORB. When developing in a CORBA environment, we have IDL interface specifications on which the IDL–to–Java compiler needs to be run. This will produce one or more Java files which have to be compiled with a Java compiler to yield Java byte code. The JacORB generator can then be run on these files in order to create the client side stub and the server side skeleton.

If, on the other hand, the interface specification is in Java, the first step of running the IDL compiler can be skipped. In this case, the instances of the generated Java classes are no CORBA objects but can still be used in distributed Java programs. The whole process of generating stubs and skeletons from either IDL or Java interfaces is illustrated in figure 3.1.

3.2.2 Concurrency

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 implementa-

[1] In future versions there will be one daemon process on every network node responsible for starting server processes.

[2] We are in fact planning to build an Interface Repository and an OA.

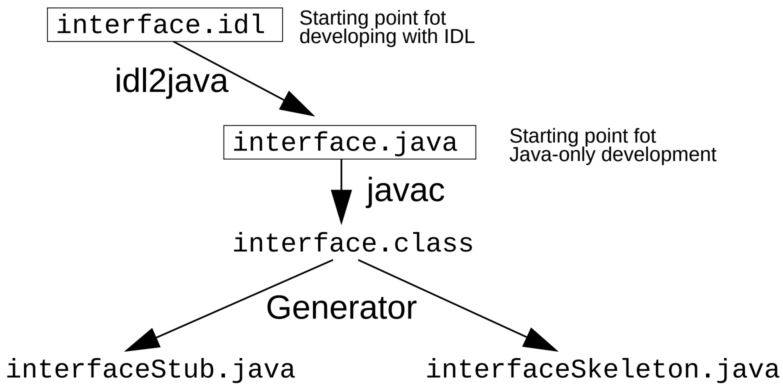

Figure 3.1: Generating Stubs

tion 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. In JacORB, the ORB core is designed so as to allow multi–threading, so no user intervention is necessary unless concurrency is to be suppressed.

JacORB Skeletons allow concurrent access to object implementations, so care must be taken to synchronize method invocations if concurrent access could potentially lead to inconsistencies or deadlock. You can disallow concurrency altogether by declaring all methods as `synchronized` in Java.

Stubs use conventional synchronous message passing semantics by default, but if an operation is declared as `oneway` in the IDL interface, the generator produces code for asynchronous message passing as well.[3]

3.2.3 Names and Object References

Using the name service, JacORB offers a simple way to locate object implementations by name. When a variable is to be bound to an object implementation, the implementation can be located by querying the name service as shown in the following code fragment:

```
server s = (server)NameServer.locate("server", context_url ));
```

The result of such a query 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. (The second argument to the call, `context_url`, is a string in URL format which is used to identify the naming context

[3] Strictly speaking, the IDL keyword `oneway` does not prescribe asynchronous invocation semantics but "best–effort" semantics, i.e. message delivery is not guaranteed. Therefore, `oneway` must not have results or `out`–Parameters. Most ORB implementations interpret this as indicative of asynchronous invocation semantics, and so do we.

which is to be used for resolving the name.)

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 would be part of the server program holding the object implementation `serverImpl`. A skeleton for this object is created and then registered.

```
serverImpl s = new serverImpl();
serverSkeleton st = new serverSkeleton( s );
NameServer.registerService( st, "server", context_url );
```

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. Stubs are only used for remote objects, and there is only one stub object in an address space for any remote object. JacORB takes care that no superfluous stubs are created or used.

3.2.4 The Java language mapping

The OMG issued an RFP for a Java language mapping in August 1996 [OMG96] to which there is a joint submission by all the major ORB vendors [Joint97]. 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.

For our purposes, we combine features from both SunSoft's and IONA's original language mappings rather than invent a complete mapping of our own here. There are, however, a number of points where our mapping differs from the proposed language mapping in [Joint97]. We will point these out in the remainder of this section. 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.

Names and Scopes

IDL provides more scoping constructs than Java does: Modules, interfaces and structs define their own name spaces. Modules are mapped to Java packages, but Java interfaces or classes do not allow further nesting of declarations for name scoping purposes.[4] In Java, we have to create new files for every public class generated from structured IDL types. These classes belong to their respective module/package, and their scope extends to the

[4] This applies to Java JDK 1.0.2. In JDK 1.1, nested classes are supported.

whole package and cannot be further constrained. The only way to avoid name clashes with the created classes is to prefix all class names with the name of the surrounding scopes if these scopes are not interfaces but, e.g., structs:

```
module Example {
   typedef char t;
   interface Alpha{
      void gr();
      struct Struct {
         sequence <t> longies;
         struct Me {
            long y;
         } you;
         char x;
}; }; };
```

results in Java classes `Alpha.java`, `AlphaStruct.java` and `AlphaStructMe.java`.

Basic Data Types

A Java `long` is a 64-bit integer, whereas an IDL `long` is only 32 bits, so IDL `long`s are mapped onto Java `int`s. Java does not support unsigned data types, so signed IDL integers map to their unsigned Java equivalents. Care has to be taken when using large values of unsigned IDL values which could result in negative values in Java. Strings can be bounded in IDL, though not in Java, so we lose information in mapping bounded strings onto Java strings.

The following mapping for basic types is used:

IDL type	Java type
`boolean`	`boolean`
`char`	`char`
`octet`	`byte`
`string`	`java.lang.String`
`short`	`short`
`unsigned short`	`short`
`long`	`int`
`unsigned long`	`int`
`float`	`float`
`double`	`double`

Enumerations

Enumerations are not supported in Java, so an IDL enum declaration produces a final Java class of the same name with `static final int` members with names corresponding to the elements of the enumeration type in IDL. This generated class implements the interface `jacorb.Orb.Enum` in order to make it possible for the runtime to distinguish between classes generated from enumerations, structures, interfaces or arrays.

```
   enum color { Red, Green, Blue, Yellow };
```

maps onto the Java class color.java:

```
public class color implements jacorb.Orb.Enum {
   public static final int Red = 0;
   public static final int Green = 1;
   public static final int Blue = 2;
   public static final int Yellow = 3;
}
```

Constants

Similarly, IDL constants are mapped onto Java classes as there is no equivalent Java language construct:

IDL:

```
   const string hallo = "hallo";
```

Java:

```
   public final class hallo {
      public static final String value = "hallo";
   }
```

The language mapping in [Joint97] proposes to map constants which are declared within an IDL interface to a `public static final` field of the corresponding Java interface. We plan to align our mapping to this proposal.

Structures

Structures are mapped onto Java final classes with public instance variables for each struct member. Two constructors are also generated, one for struct member initialization, the other empty. All classes generated from structures implement the interface `jacorb.Orb.Struct` so that the runtime system can distinguish between classes generated from structures, enumerations or interfaces.

```
   struct try_me {
      short a;
      long b;
      char c[7], d;
   };
```

gives:

```
public final class try_me       // from struct type try_me
   implements jacorb.Orb.Struct
{
   public short a;
   public int b;
   public idltest.try_mecharArray7 c;
   public char d;
   public try_me(short a, int b, idltest.try_mecharArray7 c, char d){
      this.a = a;
      this.b = b;
      this.c = c;
      this.d = d;
   }
   public try_me(){}
// serialization code...
}
```

Unions

Discriminated Unions are only hesitantly supported in our mapping as we do not regard unions as appropriate in object–oriented programming languages. The use of unions in IDL specifications is therefore strongly discouraged. Any union in IDL is treated as if it were typedef'd to the class jacorb.Idl.Union.

```
     union unionTest switch(short) {
        case 1: long l;
        default: char c;
     };
```

is equivalent to:

```
     typedef jacorb::Idl::Union unionTest;
```

This class, jacorb.Idl.Union, looks like this:

```
public class Union {
     public Object element = null;
     public Union(){}
     public Union( Object o ){
        element = o;
     }
}
```

We will, however, change our mapping to cover unions properly in the future and use the mapping for unions proposed in [Joint97].

Arrays and Sequences

Since IDL arrays are always bounded whereas Java arrays are always unbounded (in the declaration), we need to map IDL arrays to special Java classes containing fixed size arrays and implementing the interface `jacorb.Orb.Array` (for the same reasons other generated classes implement their interfaces).

```
module idltest {
   interface server {
      typedef server servers[2];
      oneway void notify(in servers svs);
   };
};
```

will therefore be mapped to a class `serverserverArray2.java`:

```
// Automatically generated by idl2java from array type servers
package idltest;
public class serverserverArray2 implements jacorb.Orb.Array {
   public idltest.server values[] = null;
   public static final int size = 2;

   public serverserverArray2(){
      values = new idltest.server[size];
   }
   public serverserverArray2( idltest.server _array[])
      throws jacorb.Idl.ArraySizeMismatchException
   {
      if( _array.length != size )
    throw new jacorb.Idl.ArraySizeMismatchException();
      values = _array;
   }
// serialization code ...
}
```

plus the interface in `server.java`:

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

public interface server extends CORBA.CORBject{
   void notify(idltest.serverserverArray2 svs);
   boolean notify$oneway = true;
}
```

Sequences are simply mapped to Java arrays. However, the mapped sequences are always unbounded (i.e. bounds information is lost in the translation process). Our mapping could be extended to produce a bounds checking operation on the generated arrays so that the original bounds of the sequence are recognized, or we could take a similar approch as with arrays. This is, however, not part of the present mapping.

```
   struct Container{
      sequence < sequence < char > > s;
      long l_array[6+3];
   };
```

is therefore mapped to:

```
public final class Container {
     // from struct type Container
     public char[][] s;
     public int l_array[];
     public Container(char[][] s, int l_array[]){
        this.s = s;
        this.l_array = l_array;
     }
     public Container() {}
}
```

Interfaces

Java supports only single implementation inheritance, but as it has multiple inheritance for interfaces, no problems arise in directly mapping inheritance clauses from IDL to Java. IDL interface attributes result in accessor methods in the Java mapping, one for retrieving the attribute value, one for setting it. If the attribute is marked `readonly`, only the `get`–method is provided.

All interfaces generated from IDL specifications extend the interface `CORBA.CORBject`. This is done in order to distinguish between such interfaces and Java interfaces not derived from IDL specifications.

IDL:

```
module jacorb {
   module Idl {
      module idltests{
         interface grid {
            attribute short height;
            readonly attribute short width;
         };
}; }; };
```

Java:

```
public interface grid extends CORBA.CORBject {
   short _get_height();
   void _set_height(short a);
   short _get_width();
}
```

Typedef

Typedefs introduce new names for IDL types. The types referred to by these newly introduced names can either be already defined, or they can be defined in the `typedef` declaration.

```
interface grid {
   typedef char harr;
   typedef sequence < harr > harrSeq;
};
```

In this example, the type name `harr` refers to a predefined type whereas `harrSeq` refers to a newly introduced type, a sequence of `harrs`. In our mapping, typedef'd names are mapped by replacing them by the mapped original type, mapping other IDL types where necessary. In the above example, it is sufficient to replace `harr` by `char`, but before `harrSeq` can be replaced, we need to map `sequence < harr >` to a Java array of `char`.

A more complete IDL example:

```
module idltest {
   interface grid {
      typedef char harr;
      typedef sequence < harr > harrSeq;
      typedef struct _h1{ harr c; } structdef;
      struct harr_test{
         harrSeq h;
      };
```

```
      void harr_func1( in harrSeq h );
      void harr_func2( in structdef s );
   };
};
```

Mapping the type name `structdef` involves mapping the structure `_h1` which generates a final class `grid_h1`:

```
// Automatically generated by idl2java
package idltest;
public final class grid_h1      // from struct type _h1
   implements jacorb.Orb.Struct
{
   public char c;
   public grid_h1(char c){
      this.c = c;
   }
   public grid_h1(){}
   // ...
}
```

The interface (in `grid.java`):

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

public interface grid extends CORBA.CORBject{
   void harr_func1(char[] h);
   void harr_func2(idltest.grid_h1 s);
}
```

Semantics of argument passing

CORBA defines pass–by–value semantics for basic (integer types, chars, etc) and constructed values (structs, sequences, arrays, unions) but pass-by-reference semantics for objects. Accordingly, JacORB stubs and skeletons marshal and un–marshal only the basic Java types (like `int, char, String`), *references* to remote objects and arrays of these kinds of objects. There is no automatic marshalling of objects of arbitrary, user-defined types. The IDL compiler includes serialization code for IDL types of non–object values in every generated Java class.

Apart from the general distinction between objects on non–object values, CORBA IDL also allows *pass-by-result* semantics (the `out` or `inout` keywords in IDL). This is modelled by having the IDL compiler automatically wrap a holder class around `inout` parameters, i.e. the objects in which we expect to receive results.

IDL:

```
interface Inout {
   void method(in long l, out short s, inout char c);
};
```

Java:

```
public interface Inout extends CORBA.CORBject{
   void method(int l,
     jacorb.Orb.IntOutHolder/*out*/ s,
     jacorb.Orb.CharHolder/*inout*/ c );
}
```

The package `jacorb.Orb` contains predefined holder classes for all basic types and generic holders for reference types. There are different holders for `out`– or `inout`–Parameters so the run time system can distinguish between these kinds of Parameters. Holders for array types will be generated by the IDL compiler when a type is used as a `out`– or `inout`–Parameter.

Any

The IDL type `any` maps to the Java class `jacorb.Orb.Any` which has methods to insert and extract values of predefined as well as of user defined types. Inserting a value sets the `typeCode` field to the appropriate value and overrides the previous contents of the `value` field. Primitive values are wrapped in appropriate object values, e.g. an `int` is wrapped in an object of the class `java.lang.Integer`.

```
package jacorb.Orb;

import CORBA.TCKind;
import CORBA.Principal;

public class Any implements Serializable {
   TypeCode typeCode;
   Object value;

   public Any(){}
   public int kind(){
      return typeCode.kind();
   }
   public void insert ( short s ){ /* ... */}
   public short extract_short() throws BadOperationException {/* ...*/}
```

```
    public void insert ( long s ) {/* ... */}
    public int extract_long() throws BadOperationException {/* ...*/}

    public void insert ( float s ) {/* ...*/}
    public float extract_float() throws BadOperationException {/* ...*/}

    // ...
    public void insert( int s, jacorb.Orb.Enum e ) throws BadOperationException {/* ...*/}
    public int extract_enum() throws BadOperationException {/* ...*/}
    // ...
}
```

Exceptions

IDL exceptions are mapped onto Java exceptions in the same manner as strutures. Declaring an exception results in the creation of a Java class of the same name which inherits from `java.lang.Exception`.

Chapter 4

JacORB Design

In this chapter we present the main components of JacORB and their design, viz. the IDL compiler, the stub generator and the JacORB runtime system.

4.1 The IDL compiler

The first thing to be noted is that the IDL–compiler is responsible for realizing the Java language mapping presented in the previous chapter. It is not responsible for generating stubs and skeletons for the Java interfaces produced during the mapping process nor for generating `TypeCode`–information. Stubs and skeletons are generated with a separate tool, taking the IDL–compiler output as its input. This tool, the JacORB generator, is described in the next section of this chapter.

Despite this decoupling, the design of this component is dependent on the classes defining the communication layer in the JacORB runtime, which will be presented below. This is due to the fact that the IDL–compiler is used to insert serialization code into generated Java classes, and this code of course depends on the communication interface.

In order to produce an IDL parser, we used a parser generator from the public domain [Hudson] which proved to be stable and reliable. It takes a LALR–Grammar for IDL written in a Lex/Yacc–like language and produces Java code for the IDL parser, relying on Java classes defining the syntactic categories for the parse tree. These Java classes are also used in the syntax tree and for the tranformation from the parse tree into the syntax tree. They also contain the methods for the Java code generation.

This design is in fact not very modular. It would be desirable to decouple the parse tree from the syntax tree in order to allow for other language mappings to be supported. As it stands, supporting other languages than Java with the same parser front–end would not be easy. A redesign of the parser classes is, however, presently beyond our resources.

All files relevant for the IDL–compiler are in the `jacorb/IDL` directory. The IDL–Grammar is in the file `parser.cup`. A few classes from the parser generator package itself are needed, and these are not part of the JacORB distribution so that it is necessary to download and install the parser generator in addition to JacORB.

4.2 The Generator

The JacORB generator generates Java code for stub and skeleton classes. It achieves this by parsing not the Java *source* code produced by the IDL–compiler, but by parsing the Java *byte code* produced by the Java compiler from the source code produced by the IDL–compiler. This approach has two advantages: First, it allows the generator to be used for distributed Java programming without IDL specifications, and second, it simplifies the IDL–compiler component by factoring out an important task.

JacORB was first designed to allow remote object invocations in Java[1] and the generator was the main tool at that time. Parsing Java byte code instead of Java source code relieves the generator from the burden of syntax checking possibly incorrect Java source code and from re–implementing existing technology. Java byte code, on the other hand, is a well documented and compact format that is much easier to parse.

4.2.1 Stubs

A stub is a client–side proxy for another object and is the concrete realization of an object reference. A client can invoke operations using this reference and pass it as an argument to other operation invocations where an object of the refernce's type was expected. Therefore, a stub must conform to the type of the object it represents. Object types are defined in IDL as interfaces, and these are mapped by the IDL–compiler to Java interfaces. The object implementation conforms to the mapped Java interface by implementing it, and so does the stub implementation. Thus, both the stub and the object implementation are subtypes of the interface defining the IDL type and both can be used in any place where an object of the interface type is expected.

The generator accordingly produces a stub class which implements the interface of the IDL type. This stub class implements all the methods listed in the original interface and also inherits a number of management operations and methods implementing the `CORBA.ORBject` interface from `jacorb.Orb.Stub`. Methods that invoke remote operations have the same signature as the interface methods and are responsible for sending operation arguments over a communication link to the remote object. They also receive and return results or exceptions from the remote operation.

As an example, consider the following IDL interface:

[1] That was before the advent of RMI, of course.

```
module jacorb {
   module demo {
      module example1 {
         interface server {
            string writeMessage(in string a1);
}; }; }; };
```

The IDL–compiler produces the following Java interface from this IDL definition:

```
package jacorb.demo.example1;
public interface server extends CORBA.CORBject{
   java.lang.String writeMessage(java.lang.String a1);
}
```

The generator finally produces the following stub class file:

```
package jacorb.demo.example1;
import jacorb.*;
public class serverStub extends jacorb.Orb.Stub implements server {
   public String typeId() {
      return "jacorb.demo.example1.server";
   }
   public java.lang.String writeMessage(java.lang.String a1) {
      Object args[] = new Object[1];
      args[0] = jacorb.Orb.External.wrapArg(a1);
      Object result = invoke("writeMessage",
               "Ljava.lang.String;)Ljava.lang.String;", args);
      return (java.lang.String)result;
   }
}
```

This example shows that JacORB stubs do not contain marshalling code for sending and receiving arguments and results, but rather wrap arguments in an `Object` array and pass them on to a generic `invoke` operation of the superclass `Stub`. This in turn uses the class `jacorb.Orb.External` for marshalling and unmarshalling.

4.2.2 Skeletons

Skeletons are the server–side equivalent of stubs. They contain code to receive operation arguments, invoke the operation on the object implementation and send any results or exceptions from the operation back to the caller. Skeletons need not conform to any supertype as they are not passed around. The following (abridged) skeleton code was generated from the above Java interface `server` and exemplifies the general skeleton layout:

```
package jacorb.demo.example1;
import java.net.*;
import java.io.*;

public class serverSkeleton extends jacorb.Orb.Skeleton {
   server objImpl;
   jacorb.Orb.Request r;
   jacorb.Orb.External e;
   // details ommitted
   public void run() {
      jacorb.Orb.ObjectInputStream o = r.in_bytes;
      try {
         if( r.mline.equals("writeMessage")) {
            java.lang.String _a0 = (java.lang.String)o.receive_object(
                                                    "Ljava.lang.String;");
            jacorb.Orb.Holder out_args[] = null;
            Object result = null;
            try{
               result = jacorb.Orb.External.wrapArg(objImpl.writeMessage(_a0));
            } catch( Exception ex ){
               if( r.response_expected )
                  e.sendReply(r, getExceptionTrace(ex), null,null,
                                        GIOP.ReplyStatusType.USER_EXCEPTION);
               return;
            }
            if( r.response_expected )
               e.sendReply(r, result, "Ljava.lang.String;", out_args,
                                        GIOP.ReplyStatusType.NO_EXCEPTION);
         }
         // further detail ommitted
         } else
            throw new jacorb.Orb.NetException("Method not found: " + r.mline);
      } catch ( Exception exc ) {
         exc.printStackTrace();
}  }  }
```

JacORB skeletons are by default multi–threaded so they inherit, through their superclass `Skeleton`, from the Java library–class `Thread` (multi–threading in JacORB is presented in more detail below). A skeleton's principal method is `run`, where dispatching of methods is performed by looking up operation names. Operation signatures are not considered, since CORBA does not allow overloading of operation names. After identifying which method is to be called, the necessary arguments are received and supplied as parameters to the actual method call. Any exceptions thrown during the method execution are caught and sent back to the caller, as are results or `out`–Parameters in case a result is expected.[2]

4.3 The JacORB Runtime

The JacORB runtime system is responsible for a number of things which will be detailed in this section. Apart from marshalling and unmarshalling data according to CORBA's interoperability requirements, the runtime system supports concurrent accesses to objects and manages object references.

4.3.1 Interoperability

CORBA specifies a protocol for interoperability between ORB implementations — the *General Inter–ORB Protocol* (GIOP). GIOP prescribes a number of message formats and an external data representation, the *Canonical Data Representation* (CDR). JacORB implements the *Internet Inter–ORB Protocol* (IIOP)[3] natively and thus interoperates with any other CORBA 2.0 compliant ORB without need for bridging.

JacORB uses the class `jacorb.Orb.External` for setting up GIOP messages and the classes `jacorb.Orb.ObjectInputStream` and `ObjectOutputStream`[4] for marshalling and unmarshalling data according to CDR. For setting up TCP connections the standard Java library sockets are used. These names of these classes `ObjectInputStream` and `ObjectOutputStream` are abbreviated as `In` and `Out` in figure 4.1.

4.3.2 Object References

The JacORB runtime ensures that there will be no more than one stub for a given remote object in a client address space in order to avoid unnecessary overhead.

4.3.3 Concurrency

As has been pointed out above, distributed systems are generally concurrent, and an ORB implementation needs to address concurrency in one form or another. In order to explain how JacORB handles concurrent accesses, we first describe the client side of object invocations and then the server side.

---

[2] CORBA allows to indicate whether results are expected in the request message header.

[3] IIOP is GIOP over TCP/IP connections.

[4] These streams are not named after the stream names in the new JDK 1.1. The JacORB class names were chosen independently and before JDK 1.1 was available.

Concurrent Clients

In order to allow concurrent activity in clients, it must be guaranteed that the runtime does not block waiting for operation replies. Only the individual client 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 threads of control. Figure 4.1 depicts this situation.

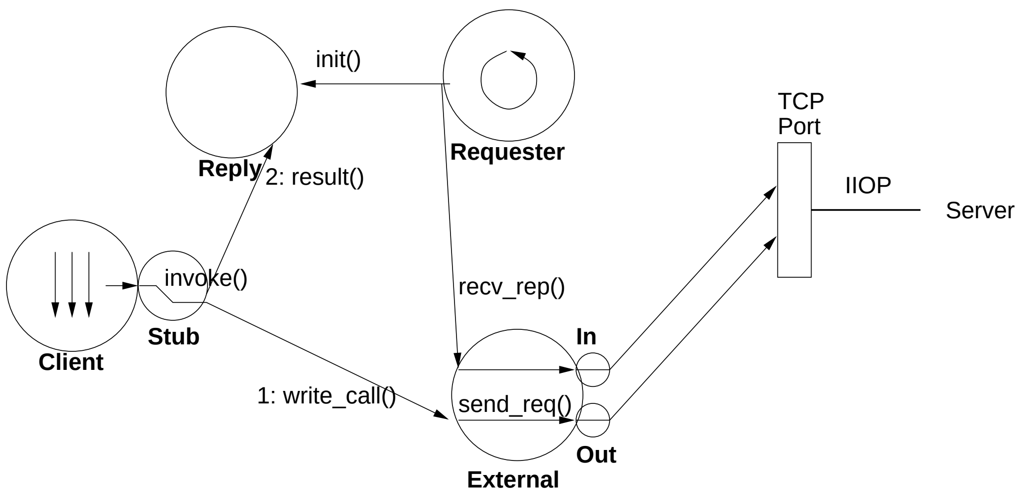

Figure 4.1: A concurrent client

A client may have any number of concurrently active threads which are allowed to invoke operations via a single object reference or stub. Access to the stub is serialized by defining a critical region with mutual exclusion around the activity of sending an individual request message (Step 1: `write_call()`). The calling thread then blocks on a `Reply` object when trying to obtain the result via the `result()`–method in Step 2. The thread is unblocked again when the `Requester`–thread receives the reply message for that `Reply`–object and notifies the waiting thread in the `init()`–method.

Concurrent Access on the Server side

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 on instance of a `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.

On order not to suppress interleaving of operation invocations from multi–threaded clients, the NetServer 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). This approach is depicted in figure 4.2. As on the client side, an `External`–object is used as the interface to the communication layer.

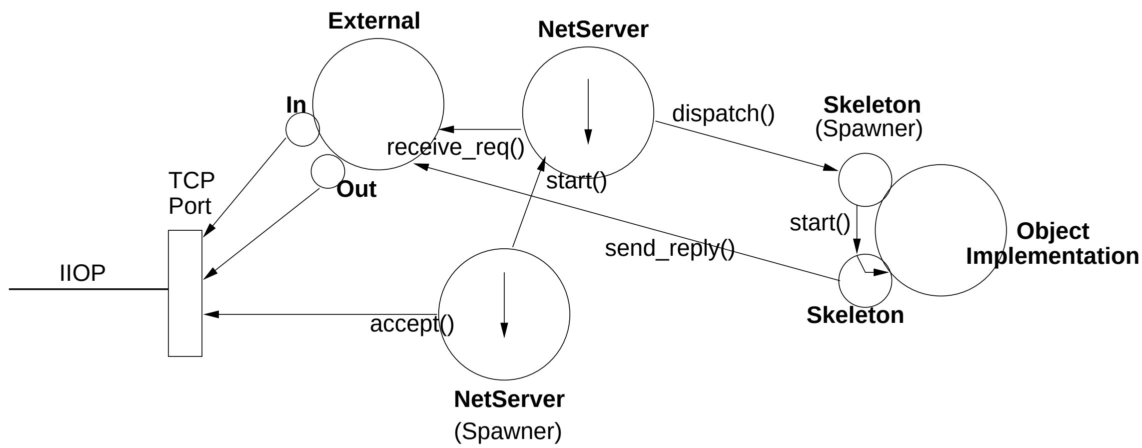

Figure 4.2: Concurrency on the server side

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.

4.3.4 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 enough meta–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.

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 `TypeCode`s 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 `TypeCode`s can thus be done by 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.

Chapter 5

Object Services

JacORB comes with an implementation of two Object Services: *Naming* and *Events*. Both these implementations should be considered prototypical rather than production quality. They are described in the following sections.

5.1 Naming

The OMG naming service allows objects to be bound to names relative to a specific name space or *naming context*. A name can be *resolved*, which means that the object bound to a given name in a naming context is determined. This service is specified in [OMG94].

The JacORB Naming Service Implementation

The JacORB naming service is an implementation of this specification. It does not, however, implement the "names library" part of the specification. The IDL interfaces from COSS 1 can be found in the file `Coss.idl` which comes with JacORB. Their Java counterparts, which were produced by using the JacORB IDL compiler, reside in the `COSS/` subdirectory. The Java interfaces for the naming service belong to the package `COSS.CosNaming` and can be found in the `COSS/CosNaming/` directory.

The central interface in the `CosNaming` package is `NamingContext`, which defines all the relevant operations on a naming context or name space. The most important operations are to bind, rebind, unbind and resolve names or contexts. The stub and skeleton generated from this interface by the JacORB generator are also part of the `CosNaming` package.

The implementation of the naming service is in the package `jacorb.Naming`. The class `jacorb.Naming.NamingContextImpl` implements the `NamingContext` interface from `COSS.CosNaming` and uses the class `Name` for operations on `NameComponents`. The API to using the name service is provided by the class `NameServer` which creates naming contexts and implements static methods to access these contexts. The relevant methods are:

```
public static jacorb.Orb.Stub locate( String service, URL u )
public static void registerService(jacorb.Orb.Skeleton s,
                     String service, URL u )
public static void unregisterService(jacorb.Orb.Skeleton s,
                     String service, URL u )
public static NamingContext getContext(URL u)
```

Starting an instance of the class `NameServer` through the class' `main` method creates one naming context.

Using the JacORB Naming Service

A name server instance managing a name context can be started by issuing the command[1]

```
$ ns filename
```

where `filename` denotes a file that can be written to by the name server. The specific name context managed by this name server can then be identified by a URL pointing to `filename` which requires that `filename` is accessible by a WWW–Server. If the local WWW–Server in a domain is, e.g., `www.inf.fu-berlin.de` and a user `brose`'s `public_html/`–directory is accessible, a name server could be started with the command

```
$ ns /home/brose/public_html/ServiceLog
```

and the context's URL would be `http://www.inf.fu-berlin.de/~brose/ServiceLog` for world–wide access or also `file:/home/brose/public_html/ServiceLog` for local access (perhaps over a shared file system).

In the JacORB naming service, different contexts are accessed through static methods of the class `jacorb.Naming.NameServer` and identified through URLs as outlined above. An object reference to a specific context can be obtained by calling the name server's static method `getContext( URL u )` with a URL identifying a name context. Using this reference, hierarchical namespaces can be configured as specified in [OMG94].

Objects can be registered with or located via the name service be first obtaining an object reference for a specific context and then calling the `bind` or `resolve` operations on that context object. The class `jacorb.Naming.NameServer` also offers static methods `locate` and `register` which take a URL as a parameter to identify the context the object is to be registered or located in. As an example, consider the following code to locate an object:

```
// File: jacorb/demo/example1/Client.java
package jacorb.demo.example1;
import jacorb.*;
import jacorb.Naming.NameServer;
import java.net.*;
```

[1] Assuming JacORB is installed properly.

```
public class Client{
   public static void main( String[] args ){
      try{
          server s = (server)NameServer.locate("server",
             new URL("http://www.inf.fu-berlin.de/~brose/ServiceLog"));
          System.out.println( s.writeMessage( args[0] ));
      } catch ( Exception e){
          e.printStackTrace();
      }
   }
}
```

For the `locate` to succeed, a name server needs to be running which has been started as outlined above to manage the name contex `http://www.inf.fu-berlin.de/~brose/ServiceLog`. Also, an object of type `server` must have been registered in this context under the name "server". If this is the case, the `locate`–operation will return an object reference to the object.

5.2 Events

The OMG Event Service is a means of decoupling objects sending and receiving asynchronous messages or events — neither suppliers nor consumers need to know the other side. It is possible to have any number of consumers of events connected to a single supplier, to have a single consumer receive events from a number of suppliers or to have a n–m relation between suppliers and consumers of events.

The events service supports two different models of event delivery: event consumers can "pull" events, i.e. actively request events, or they can be notified of events ("push"). Similarly, suppliers can generate events whenever they wish to ("push"), or they can be asked for events ("pull").

The central abstraction for the event service is that of an *event channel*. Both suppliers and consumers connect to the event channel and interact only with the channel, not with each other. Multiple channels can be configured such as to form chains of event channels where one channel acts as a consumer or supplier to another. Figure 5.1 illustrates that the event channel functions as a consumer to supplier objects and as a supplier to consumers.

Actually, it is not the event channel interface that clients — both consumers and suplliers of events — interact with, but *proxy* suppliers or consumers. A push supplier, e.g., would not invoke push operations on the event channel interface, but on a `ProxyPushConsumer`–interface to which it would have to obtain a reference first.

The typical way of interaction with an event channel, then, is to obtain a reference to a supplier or consumer "admin" interface by calling `for_suppliers()` or `for_consumers` on the event channel. The interfaces `SupplierAdmin` or `ConsumerAdmin` list operations to

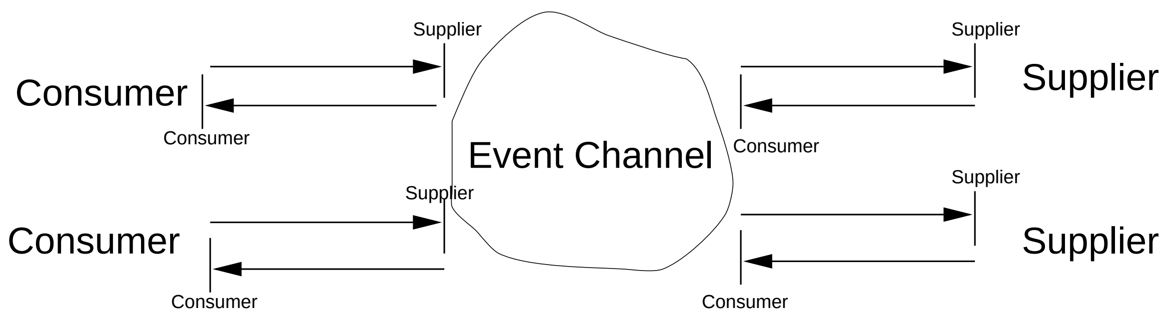

Figure 5.1: An event channel

get references to `ProxyPushConsumers`, `ProxyPullConsumers`, `ProxyPullSuppliers` and `ProxyPushSuppliers` for all four possible communication models. After obtaining one of these by calling the appropriate operation, e.g. `obtain_pull_consumer()`, the event service user has to connect to that proxy, in this case by calling `connect_pull_supplier()`.

Event communication is untyped, i.e. event data sent across a channel has the IDL type `any`, but typed channels can be used to extend the basic event communication functionality. There are no a priori quality of service requirements for event channels, so a broad range of possible implementations is conceivable.

The JacORB Event Service Implementation

The JacORB event service was implemented by Rainer Lischetzki and Jörg von Frantzius as part of their coursework for an advanced programming course in distributed object systems during the winter term 1996/1997 at Freie Universität Berlin. The implementation was later adapted to more recent JacORB versions by the author of this report.

The service implements the interfaces from the COSS event service specification. The generated Java interfaces for the event service belong to the packages `COSS.CosEventComm` and `COSS.CosEventChannelAdmin` and reside in the respective directories.

The event channel implementation is in the class `jacorb.Events.EventChannelImpl`. This class implements the two interfaces `COSS.CosEventChannelAdmin.SupplierAdmin` and `COSS.CosEventChannelAdmin.ConsumerAdmin` as well as the event channel interface itself, so the event channel object can control how many proxies it delivers to clients of the service if it wishes to.

Push style communication is straightforward. A separate thread is started after an event has been pushed into the channel so that push suppliers return immediately and do not wait until the event has been delivered. Event delivery is based on current connections at the time an event is communicated to the channel: only those consumers connected at that time will receive the event. There is no buffering of events for late–comers.

The only kind of buffering done by the channel is for pull style consumers: a linked list is kept that contains all events still waiting to be pulled out by consumers. An event is only removed from this list after all pull consumers connected at the time of the arrival of that event at the channel have called `pull()` to retrieve it. If there are no events pending when pull consumers call `pull()`, the channel tries to pull any pull–style suppliers of events in order to satisfy the consumers. It is at this time only that pull style suppliers are polled by the channel. Thus, if only pull suppliers and push consumers are connected to an event channel, no events will be communicated.

Examples for all four event communication styles can be found in the directory `jacorb.demo.Events`.

Chapter 6

Future work

This chapter gives a short overview about the directions in which JacORB is likely to develop. We first list limitations of the current release and go on to sketch our goals for further development.

6.1 Limitations

- IDL unions are not properly supported.
- We do not presently support little–endian byte order in GIOP messages.
- If servers run out of file descriptors for connections, no new connections to this server are possible. A proper connection management on the server side is needed.
- Multidimensional arrays and sequences are not handled correctly.

6.2 Further Development

Object Adapter and Implementation Repository

One of the main goals for future JacORB releases are to develop a *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.

Interface Repository

A runtime type management facility, the Interface Repository, is prescribed by CORBA. The JacORB Interface Repository will most likely be realized as one extra process for an ORB domain. The IR could make use of Java's reflection features.

Further Object Services

Further CORBA object services can be added to JacORB incrementally, e.g. Life Cycle, Concurrency Control, Transactions, Trading, Relationship, etc. A few services have been implemented as prototypes by undergraduate students: Concurrency, Persistence, Relationship. Some of these might be included in future JacORB releases though we do not regard this as essential.

CORBA compliance

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 client and server interface to the ORB itself does not presently conform to the CORBA standard where operations such as `ORB::init()`, `BOA::init()` and `ORB::list_initial_services()` are specified. We regard this as neither particularly urgent nor as problematic, though.

Bibliography

[Brose97] Gerald Brose, *An Introduction to Programming with JacORB*, http://www.inf.fu-berlin.de/~brose/jacorb/tutorial.ps.

[BB97] Gerald Brose, Boris Bokowski, "JacORB — Ein Object Request Broker für Java", to appear in: *Informatik/Informatique*, Java–Sonderheft, Juni 1997.

[GJS96] Gosling, James and Bill Joy, Guy Steele, *The Java Language Specification*, Addison–Wesley 1996.

[Hudson] Scott Hudson, *Constructor of Useful Parsers — CUP*, Version 0.9, http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.html.

[IONA96] IONA Technologies Ltd., *The Orbix Architecture*, IONA, November 1996.

[Joint97] DEC, Expersoft, HP, IBM, Netscape, Novell, Oracle, Sun, Visigenic, Xerox, *IDL/Java Language Mapping, Joint Revised Submission*, OMG TC document orbos/97-02-01, 17 February 1997.

[OHE96] Orfali, R. and D. Harkey, J. Edwards, *The Essential Distributed Objects Survival Guide*, Wiley 1996.

[OMG94] OMG , *Common Object Services Specification, Volume I*, Revision 1.0, March 1994, OMG Document 94–1–1.

[OMG95] OMG, *The Common Object Request Broker: Architecture and Specification*, 2.0, July 1995.

[OMG96] OMG, *Java Language Mapping RFP*, OMG TC document orbos/96-08-01, 1 August 1996.

[Siegel96] Jon Siegel (ed.), *CORBA Fundamentals and Programming*, Wiley 1996.

[Sun97] Sun Microsystems, Inc., *Solaris NEO*, White Paper, January 1997, http://www.sun.com/solaris/neo/whitepapers/solaris_neo/index.html.