JacORB 1.0 Programming Guide

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

Revision : 1.17

April 4, 2000

Contents

1 Introduction **3**

1.1 Status of this document 3

2 Installing JacORB **4**

2.1 Obtaining JacORB 4

2.2 Installation 4

3 Getting Started **8**

3.1 JacORB development: an overview 8

3.2 IDL specifications 8

3.3 Generating a Java classes 9

3.4 Implementing the interface 10

3.5 Writing the Server 11

3.6 Writing a client 13

4 The JacORB Name Service **15**

4.1 Running the Name Server 15

4.2 Accessing the Name Service 16

4.3 Constructing Hierarchies of Name Spaces 17

4.4 NameManager — A simple GUI front-end to the Naming Service 18

5 The server side: POA, Threads **19**

5.1 POA 19

5.2 Threads 19

6 Implementation Repository **21**

6.1 Overview 21
6.2 Using the JacORB Implementation Repository 22
6.3 Server migration 24

7 Interceptors 25

7.1 Introducing Interceptors 25
7.2 Request–level Interceptors 26
7.2.1 Client–side interceptors 27
7.2.2 Server–side interceptors 28
7.2.3 Per–object vs. per–process Interceptors 28
7.2.4 Registering Interceptors 28
7.3 Message–level Interceptors 29
7.3.1 Example: A simple Encryption Interceptor 30
7.3.2 Per–object vs. per–process Interceptors 31

8 Applets — The JacORB Appligator 33

8.1 Using Appligator 33

9 IIOP over SSL 36

9.1 Abstract 36
9.2 Required 36
9.3 Introduction 36
9.4 Building JacORB with SSL support 37
9.5 Creating key stores 37

10 JacORB utilities 39

10.1 idl 39
10.2 ns 40
10.3 nmg 41
10.4 lsns 41
10.5 dior 42
10.6 pingo 42

11 Limitations, Feedback 43

Chapter 1

Introduction

This document gives an introduction to programming distributed applications with JacORB, a free Java object request broker. JacORB comes with full source code and a number of example programs. This document is *not* an introduction to CORBA in general. Please see [Sie96, Vin97, VD98] for this purpose and [HV99] for more advanced issues. The JacORB version described in this document is JacORB 1.1.

1.1 Status of this document

This document is very much in a preliminary state. A number of things have changed from version 0.9, others — like the interface repository — are unavailable at the moment but will reappear in future releases. The most prominent changes that a programmer will notice are in the language mapping, i.e. the way the IDL compiler generated Java classes from IDL specifications.

In previous releases, JacORB used a somewhat idiosyncratic mapping in order to allow for a special design of the Interface Repository[Bro98] and, in particular, for CORBA programming without any hand–written IDL. All that had to give way to the POA. Supporting the Portable Object Adapter meant adopting the standard OMG IDL language mapping [OMG98], which in turn meant redesigning quite a bit of the IDL compiler.

This document is intended to give a few essential hints on how to install and use JacORB, but it will not suffice as either a gentle introduction to CORBA or a tutorial. The rest of this document is structured as follows. First, we briefly describe how to obtain and install JacORB. Section 2 gives a few examples on how to use JacORB to write distributed Java programs while section 3 contains a description of the utilities that come with JacORB.

Chapter 2

Installing JacORB

In this chapter we explain how to obtain and install JacORB and give an overview of the package contents.

2.1 Obtaining JacORB

JacORB can be obtained as a g-zipped tar–archive or as a zip–archive from the JacORB home page at `http://www.inf.fu-berlin.de/~brose/jacorb/`. It can also be downloaded via anonymous ftp from `ftp.inf.fu-berlin.de` from the directory `pub/jacorb/`.

To install JacORB, just gun-zip and untar (or simply unzip) the archive somewhere. This will result in a new directory `JacORB1_1`. Make sure your CLASSPATH environment variable contains `JacORB1_1/lib/jacorb.jar`. If you plan to recompile all or parts of JacORB, you should also include `JacORB1_1/classes`. Extend your search path with `JacORB1_1/bin`, so that the shell scripts and batch files for the utilities in this directory are found.

2.2 Installation

Ant and make files

JacORB will run on any JavaVM, but to rebuild JacORB (and compile the examples) you need to have the XML–based make tool "Ant" installed on your machine. Ant can be downloaded from `http://jakarta.apache.org/ant`. All make files (`build.xml`) are written for this tool. To rebuild JacORB completely, just type `ant` in the installation directory. Optionally, you might want to do a `ant clean` first.

In order to be able to use GUI tools such as the NameManager, the ImRManager and the KeyStoreManager, you need to use JDK 1.2 or have Sun's Swing classes installed on your machine. In the latter case, your CLASSPATH also needs to contain these classes. For SSL, you must have JDK 1.2. Also, you need to install cryptography and SSL libraries by IAIK, viz. IAIK–JCE 2.6 beta 1 and iSaSiLk 3.0. Please see `http://jcewww.iaik.tu-graz.ac.at/`.

Configuration

JacORB has a number of configuration options which can be set as Java properties. Before we will go about explaining some of the more basic options here, let's look at the different ways of setting properties. Specific options that apply, e.g., to the Implementation Repository or the Trading Service are explained in the chapter discussing these moduleThere are three options for doing this.

The most general case is in a properties file. JacORB looks for and loads a file called either `.jacorb_properties` or `jacorb.properties`. It looks for these files in both your home directory (as in the `user.home` system property and in the current directory. (The "home" on Windows95 is usually `c:\windows` or `c:\winNT\Profiles\username` on Windows NT.) If it finds multiple files, it loads the all in this order. In case of different settings for the same property, the one loaded last takes precedence.

For more application–specific properties, you can pass a `java.util.Property` object to `ORB.init()` during application initialization, e.g. like this (args is the String array containing command line arguments):

```
    java.util.Properties props = new java.util.Properties();
    props.setProperty("jacorb.implname","StandardNS");
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);
```

The third way of specifying properties is by passing them as command line arguments to the Java interpreter, as in:

```
  $ java -DOAPort=4711 jacorb.naming.NameServer ...
```

An option passed like this will override any setting in either the properties file or the application setup. Properties set in the application code, however, will always override any properties set in any other way.

We are now ready to have a look at the most basic JacORB configuration properties. Here is an example `jacorb_properties` file:

```
##
##  JacORB configuration options
##

#
# Basic ORB properties
#

# number of retries if connection cannot directly be established
```

```
jacorb.retries=5
jacorb.retry_interval=700

# size of network buffers for outgoing messages
jacorb.outbuf_size=2048

# URL where IORs are stored (used in orb.resolve_initial_service())
jacorb.NameServerURL=http://www.inf.fu-berlin.de/˜brose/NS_Ref
jacorb.TradingServiceURL=http://www.inf.fu-berlin.de/˜brose/TS_Ref
jacorb.ImplementationRepositoryURL=http://www.inf.fu-
berlin.de/˜brose/ImR_Ref
jacorb.ProxyServerURL=http://www.inf.fu-berlin.de/˜brose/Proxy_Ref

# amount of diagnostic output, 0 is off
jacorb.verbosity=1

# where does output go? Terminal is default
#jacorb.logfile=LOGFILEPATH

# Switch off to avoid contacting the ImR if not used, must be on
# to be able to use it.
jacorb.use_imr=on

# default implementation name for servers
# with persistent POAs
jacorb.implname=StandardImplName

#
# interceptor configuration, interceptors can be
# configured out for performance
#
jacorb.interceptor.client.Requests=on
jacorb.interceptor.client.Messages=on
jacorb.interceptor.server.Messages=on
jacorb.interceptor.server.Requests=on
jacorb.interceptor.perObject=on

#
# POA configuration
#

# displays a GUI monitoring tool for servers
jacorb.poa.monitoring=off

# thread pool configuration for request processing
jacorb.poa.thread_pool_max=20
jacorb.poa.thread_pool_min=10

# size of the request queue, clients will receive Corba.TRANSIENT
# exceptions if load exceeds this limit
jacorb.poa.queue_max=100

#
```

```
# Naming Service Configuration
#
# where to store naming context state?
jacorb.naming.db_dir=/home/bwana/brose/tmp

#
# Implementation Repository Configuration Options
#
...

#
# Trader configuration, please see
# src/trading/README.PROPERTIES for
# explanation
...
```

Configurable options include the size of network buffers, the number of retries JacORB makes if a connection cannot be established, and how long it shall wait before retrying. The string values for `NameServerURL` and `TradingServiceURL` are URLs for file resources used to set up the JacORB name server. This URL will be used by the ORB to locate the name server and to provide an object reference to it (see also chapter 4).

The `verbosity` option tells JacORB how much diagnostic output it should emit at runtime. Unless the `logfile` property is set to a file name, diagnostic output wil be sent to the terminal. Setting the verbosity property to 0 means don't print any, while a level of 2 is a verbose debug mode. Level 1 will emit some information, e.g. about connections being opened, accepted and closed.

The interceptor section in the properties file determines which interceptors are used. By default, all interceptors are switched off for performance reasons. Please see chapter 7 for details about interceptors.

The `monitoring` option determines whether the POA should bring up a monitoring GUI for servers that let you examine the dynamic behaviour of your POA, e.g. how long the request queue gets and whether your thread pool is big enough. Also, this tool lets you change the state of a POA, e.g. from *active* to *holding*. [Please see the chapter on the POA for more details.]

You can now test your installation by typing `make` in one of the subdirectories of the `demo/` directory which contains a number of examples for using JacORB.

Chapter 3

Getting Started

Before we go about explaining an example in detail, we will have a look at the general process of developing CORBA applications with JacORB. We'll follow this roadmap when working through the example. The example can be found in `demo/grid` which also contains a Makefile so that the development steps do not have to be carried out manually every time. Still, you should know what is going on.

As this document gives only a short introduction to JacORB programming and does not cover all the details of CORBA IDL, we recommend that you also look at the other examples in the `demo/` directory. These are organized so as to show how the different aspects of CORBA IDL can be used with JacORB.

3.1 JacORB development: an overview

The steps we will generally have to take are:

1. write an IDL specification.
2. compile this specification with the IDL compiler to generate Java classes.
3. write an implementation for the interface generated in step 2
4. write a "Main" class that instantiates the server implementation and registers it with the ORB
5. write a client class that retrieves a reference to the server object.

3.2 IDL specifications

Our example uses a simple server the definition of which should be clear if you know IDL. Its interface is given in `server.idl`. All the source code for this example can be found in `JacORB1_0/demo/grid`.

```
// server.idl
// IDL defintion of a 2-D grid:

module grid
{
    interface MyServer
    {
        readonly attribute short height;  // height of the grid
        readonly attribute short width;   // width of the grid

        // set the element [n,m] of the grid, to value:
        void set(in short n, in short m, in long value);

        // return element [n,m] of the grid:
        long get(in short n, in short m);

        exception MyException
        {
            string why;
        };

        short opWithException() raises( MyException );
    };
};
```

3.3 Generating a Java classes

Feeding this file into the IDL compiler

```
$ idl -d ../..  -p demo server.idl
```

produces a number of Java classes that represent the IDL definitions. This is done according to a set of rules known as the IDL–to–Java language mapping as standardized by the OMG. If you are interested in the details of the language mapping, i.e. which IDL language construct is mapped to which Java language construct, please consult the specifications available from `www.omg.org`. The language mapping used by the JacORB IDL compiler is the one defined in CORBA 2.3 and is not explained in detail in this document. For practical usage, please consult the examples in the `demo` directory.

The most important Java classes generated by the IDL compiler are the interfaces `MyServer` and `MyServerOperations` plus the stub and skeleton files `_MyServerStub`, `MyServerPOA` and `MyServerPOATie`. We will use these classes in the client and server as well as in the implementation of the grid's functionality and explain each in turn.

Note that the IDL compiler will produce a directory structure for the generated code that corresponds to the module structure in the IDL file, so it would have produced a subdirectory `demo/grid` in the current directory had we not directed it to put this directory structure to `../..` by using the compiler's `-d` switch. The enclosing module `demo` is generated by applying the `-p demo` option which instructs the compiler to behave as if the IDL file had been written with an outermost module `demo`. As a result, all generated files that would have been in the `demo/grid` subdirectory are now put into the current directory — which is already called `demo/grid`. Where to put the source files for generated classes is a matter of taste. Some people prefer to have everything in one place (as using the `-d` option in this way achieves), others like to have one subdirectory for the generated source code and another for the output of the Java compiler, i.e. for the `.class` files.

3.4 Implementing the interface

Let's try to actually provide an implementation of the functionality promised by the interface. The class which implements that interface is called `gridImpl`. Apart from providing a Java implementation for the operations listed in the IDL interface, it has to inherit from a generated class that both defines the Java type that represents the IDL type `MyServer` and contains the code needed to receive remote invocations and return results to remote callers. This class is `MyServerPOA`.

You might have noticed that this approach is impractical in situations where your implementation class needs to inherit from other classes. As Java only has single inheritance for implementations, you would have to use an alternative approach — the "tie"–approach — here. The tie approach will be explained later.

Here is the Java code for the grid implementation:

```
package demo.grid;

import demo.grid.MyServerPackage.MyException;

public class gridImpl
    extends MyServerPOA
{
    protected short height = 31;
    protected short width = 14;
    protected int[][] mygrid;

    public gridImpl()
    {
        mygrid = new int[height][width];
        for( short h = 0; h < height; h++ )
           for( short w = 0; w < width; w++ )
               mygrid[h][w] = 0;
    }

    public int get(short n, short m)
    {
        if( ( n <= height ) && ( m <= width ) )
            return mygrid[n][m];
        else
            return 0;
    }
    public short height()
    {
        return height;
    }

    public void set(short n, short m, int value)
    {
        if( ( n <= height ) && ( m <= width ) )
            mygrid[n][m] = value;
    }

    public short width()
    {
        return width;
    }

    public short opWithException()
        throws MyException
    {
        throw new MyException(‘‘This is only a test excep-
tion, no harm done :-)’’);
    }
}
```

3.5 Writing the Server

To actually instantiate a `gridImpl` object which can be accessed remotely as a CORBA object of type `MyServer`, you have to instantiate it in a main method of some other class and register it with a component of the CORBA architecture known as the *Object Adapter*. Here is the class `Server` which does all that is necessary to activate a CORBA object of type `MyServer` from a Java `gridImpl` object:

```
package demo.grid;

import java.io.*;
import org.omg.CosNaming.*;

public class Server
{
    public static void main( String[] args )
    {
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
        try
        {
            org.omg.PortableServer.POA poa =
                org.omg.PortableServer.POAHelper.narrow(
                    orb.resolve_initial_references("RootPOA"));

            poa.the_POAManager().activate();

            org.omg.CORBA.Object o = poa.servant_to_reference(new gridImpl());

            if( args.length == 1 )
            {
                // write the object reference to args[0]

                PrintWriter ps = new PrintWriter(
                                    new FileOutputStream(
                                       new File( args[0] )));
                ps.println( orb.object_to_string( o ) );
                ps.close();
            }
            else
            {
                // register with the naming service

                NamingContextExt nc =
                     NamingContextExtHelper.narrow(
                        orb.resolve_initial_references("NameService"));
                nc.bind( nc.to_name("grid.example"), o);
            }
        }
        catch ( Exception e )
        {
            e.printStackTrace();
        }
        orb.run();
    }
}
```

After initializing the ORB we need to obtain a reference to the object adapter — the POA — by asking the ORB for it. The ORB knows about a few initial references that can be retrieved using simple names like “RootPOA’. The returned object is an untyped reference of type `CORBA.Object` and thus needs to be narrowed to the correct type using a static method `narrow()` in the helper class for the type in question. We now have to activate the POA because any POA is created in “holding” state in which it does not process incoming requests. After calling `activate()` on the POA’s POAManager object, the POA is in an active state and can now be asked to create a CORBA object reference from a Java object also know as a `Servant`.

In order to make the newly created CORBA object accessible, we have to make its object reference available. This is done using a publicly accessible directory service, the naming server. A reference to the naming service is obtained by calling `orb.resolve_initial_references("NameService")` on the ORB and narrowing the reference using the `narrow()` method found in class `org.omg.CosNaming.NamingContextExtHelper`. Having done this, you should call the `bind()` operation on the name server. The name for the object which has to be supplied as an argument to `bind()` is not simply a string. Rather, you need to provide a sequence of `CosNaming.NameComponent`s that represent the name. In the example, we chose to use an extended Name Server interface that provides us with a more convenient conversion operation from strings to Names.

3.6 Writing a client

Finally, let's have a look at the client class which invokes the server operations:

```
package demo.grid;

import org.omg.CosNaming.*;

public class Client
{
    public static void main(String args[])
    {
        try
        {
            MyServer grid;
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

            if(args.length==1 )
            {
                // args[0] is an IOR-string
                grid = MyServerHelper.narrow(orb.string_to_object(args[0]));
            }
            else
            {
                NamingContextExt nc =
                         NamingContextExtHelper.narrow(
                                orb.resolve_initial_references("NameService"));

                grid = MyServerHelper.narrow(
                                nc.resolve(nc.to_name("grid.example")));
            }

            System.out.println("Height = " + grid.height());
            System.out.println("Width = " + grid.width());

            try
            {
                grid.opWithException();
            }
            catch (jacorb.demo.grid.MyServerPackage.MyException ex)
            {
                System.out.println("MyException, reason: " + ex.why);
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

After initializing the ORB, the client obtains a reference to the "grid" service by locating the reference using the name service. Again, resolving the name is done by getting a reference to the naming service by calling `orb.resolve_initial_references("NameService")` and querying the name server for the `"grid"` object by calling `resolve()`. The argument to the resolve operation is, again, a string that is converted to a Name. The result is an object reference of type `org.omg.CORBA.Object` which has to be narrowed to the type we are expecting, i.e. `MyServer`.

After compiling everything we're now ready to actually run the server and the client on different (virtual) machines. Make sure the name server is running before starting either the server or the client. If it isn't, type something like:

```
$ ns ~/public_html/NS_Ref
```

where `~/public_html/NS_Ref` is the name of a locally writable file which can be read by using the URL given in both the remote client and server code. (This is to avoid using a well–known address for the name server, so both client and server look up the location of the nameserver via the URL and later communicate with it directly.)

You can now launch the server:

```
$ jaco demo.grid.Server
```

The client can be invoked on any machine you like:

```
$ jaco demo.grid.Client
```

Running the client after starting the server produces the following output on your terminal:

```
Height = 31
Width = 14
MyException, reason: This is only a test exception, no harm done :-)
```

Chapter 4

The JacORB Name Service

Name servers are used to locate objects using a human–readable reference (their name) rather than a machine or network address. If objects providing a certain service are looked up using the service name, their clients are decoupled from the actual locations of the objects that provide this service. The binding from name to service can be changed without the clients needing to know.

JacORB provides an implementation of the OMG's Interoperable Naming Service which allows to bind names to object references and to lookup object references using these names. It also allows to easily convert names to strings and vice versa The JacORB name service comprises two components: the name server program, and a set of interfaces and classes used to access the service.

One word of caution about using JDK 1.2 with the JacORB naming service: JDK 1.2 comes with a couple of outdated and apparently buggy naming service classes that fo not work properly with JacORB. To avoid having these classes loaded and used inadvertently, please make sure that you always use the `NamingContextExt` interface rather than the plain `NamingContext` interface. Otherwise, you will see your application receive null pointer or other exceptions. Note that there is no such problem with JDK 1.1.

4.1 Running the Name Server

The JacORB name server is a process that needs to be started before the name service can be accessed by programs. Starting the name server is done by typing on the command line either simply

```
ns <ior filename> [<timeout>]
```

to run a shell script or

```
ns.bat <filename> [<timeout>]
```

for a DOS batch file. You can also start the Java interpreter explicitly by typing

```
jaco jacorb.Naming.NameServer <filename> [<timeout>]
```

In the example

```
ns ~/public_html/NS_Ref
```

we direct the name server process to write location information and logging information to the file

`~/public_html/NS_Ref`. A client–side ORB uses this file to locate the name server process. The client–side ORB does not, however, access the file through a local or shared file system by default. Rather, the file is read as a WWW resource by using a URL pointing to it. This implies that the name server log file is accessible through a URL in the first place, i.e. that you know of a web server in your domain which can answer HTTP request to read the file.

The advantage of this approach is that clients do not need to rely on a hard–coded well known port and that the name server is immediately available world–wide if the URL uses HTTP. If you want to restrict name server visibility to your domain (assuming that the log file is on a shared file system accessible throughout your domain) or you do not have access to a web server, you can use file URLs rather than HTTP URLs, i.e. the URL pointing to your name server log file would looks like

`file:/home/brose/public_html/NS_Ref`

rather than

`http://www.inf.fu-berlin.de/~brose/NS_Ref`

Specifying file URLs is also useful on machines that have no network connection at all. Please note that the overhead of using HTTP is only incurred once — when the clients first locate the name server. Subsequent requests will use standard CORBA operation invocations which means they will be IIOP requests (over TCP).

The name server stores its internal state, i.e. the name bindings in its context, in files in the current directory unless the property `jacorb.naming.db_dir` is set to a different directory name. This saving is done when the server goes down regularly, i.e. killing the server with CTRL-C will result in loss of data. The server will restore state from its files if any files exist and are non–empty.

The second parameter is a time–out value in msecs. If this value is set, the name server will shut down after the specified amount of time and save its state. This is useful if the name server is registered with the Implementation Repository and can thus be restarted on demand.

Configuring a Default Context

Configuring a naming context (i.e. a name server) as the ORB's default or root context is done by simply writing the URL that points to this server's bootstrap file to the properties file `.jacorb_properties`. Alternatively, you can set this file name in the property `jacorb.TradingServiceURL` either on the command line or within the application as described in 2.2. After the default context has thus been configured, all operations on the NamingContextExt object that was retrieved by a call to `orb.resolve_initial_references(‘‘NameService’’)` will go to that server — provided it's running or can be started using the Implementation Repository.

4.2 Accessing the Name Service

The JacORB name service is accessed using the standard CORBA defined interface:

```
    // get a reference to the naming service
    ORB orb = ORB.init(args, null);
    org.omg.CORBA.Object o = orb.resolve_initial_references("NameService")
```

```
   NamingContextExt nc = NamingContextExtHelper.narrow( o );

   // look up an object
   server s = serverHelper.narrow( nc.resolve(nc.to_name(‘‘server.service’’)) );
```

Before an object can be looked up, you need a reference to the ORB's name service. The standard way of obtaining this reference is to call `orb.resolve_initial_references("NameService")`. In calls using the standard, extended name service interface, object names are represented as arrays of `NameComponent`s rather than as strings in order to allow for structured names. Therefore, you have to construct such an array and specify that the name's name is ”server” and that it is of kind ”service” (rather than ”context”). Alternatively, you can convert a string “server.service” to a name by calling the `NamingContextExt` interface's `to_name()` operation, as shown above.

Now, we can look up the object by calling `resolve()` on the naming context, supplying the array as an argument.

4.3 Constructing Hierarchies of Name Spaces

Like directories in a file system, name spaces or contexts can contain other contexts to allow hierarchical structuring instead of a simple flat name space. The components of a structured name for an object thus form a path of names, with the innermost name space directly containing the name binding for the object. This can very easily be done using `NameManager` but can also be explicitly coded.

A new naming context within an enclosing context can be created using either `new_context()` or `bind_new_context()`. The following code snippet requests a naming context to create an inner or subcontext using a given name and return a reference to it:

```
   // get a reference to the naming service
   ORB orb = ORB.init();
   org.omg.CORBA.Object o = orb.resolve_initial_references("NameService")
   NamingContextExt rootContext = NamingContextExtHelper.narrow( o );

   // look up an object
   NameComponent[] name = new NameComponent[1];
   components[0] = new NameComponent("sub","context");
   NamingContextExt subContext = NamingContextExtHelper.narrow( root-
Context.bind_new_context( name ));
```

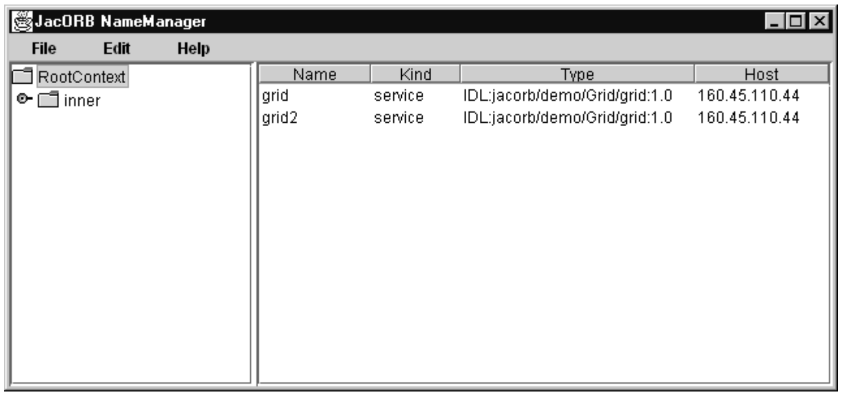
Please note that the JacORB naming service always use `NamingContextExt` objects internally, even if the operation signature indicates NamingContext objects. This is necessary because of the limitations with JDK 1.2 as explained at the beginning of this section.

4.4 NameManager — A simple GUI front-end to the Naming Service

Provided that you are using JDK 1.2 or have the JFC or swing classes installed properly on your machine, the graphical front-end to the name service can be started by simply calling

```
nmg
```

The GUI front-end will simply look up the default context and display its contents. Here is a screen shot:



Name	Kind	Type	Host
grid	service	IDL:jacorb/demo/Grid/grid:1.0	160.45.110.44
grid2	service	IDL:jacorb/demo/Grid/grid:1.0	160.45.110.44

NameManager has menus that let you unbind names and create or delete naming contexts within the root context. Creating a nested name space, e.g., can be done by selecting the `RootContext` and bringing up a context by clicking the right mouse button. After selecting “new context”from that menu, you will be prompted to enter a name for the new, nested context.

Chapter 5

The server side: POA, Threads

This chapter describes the facilities offered by JacORB for controlling how servers are started and executed. These include an activation daemon[1], the Portable Object Adapter (POA) and its interface and threading.

Presently, this chapter is largely unfinished and a mere placeholder. A more thorough explanation of how the POA can be used in different settings and how the different policies and strategies it offers are best used is on our to–do list. For now, we have to refer you to the excellent chapter on the POA in [HV99]. A very quick overview of the POA can be found in [Vin98] while more in–depth treatment (albeit in C++) can be found in the various C++-Report Columns on the POA by Doug Schmidt and Steve Vinoski. These articles are available at `http://www.cs.wustl.edu/˜schmidt/report-doc.html`.

5.1 POA

The object adapter is that part of CORBA that is responsible for creating CORBA objects and object references and — with a little help from skeletons — dispatching operation requests to actual object implementations. In cooperation with the Implementation Repository it can also activate objects, i.e. start processes with programs that provide implementations for CORBA objects.

[...]

5.2 Threads

JacORB currentl offers one server–side thread model. The POA responsible for a given request will obtain a request processor thread from a central thread pool. The pool has a certain size which is always between the maximum and minimum value configure by setting the properties `jacorb.poa.thread_pool_max` and `jacorb.poa.thread_pool_min`.

When a request arrives and the pool is found to contain no threads because all existing threads are active, new threads may be started until the total number of threads reaches

[1]not in 1.0.

`jacorb.poa.thread_pool_max`. Otherwise, request processing is blocked until a thread is returned to the pool. Upon returning threads that hav finished processing a request to the pool, it must be decided whether the thread should actually remain in the pool or be destroyed. If the current pool size is above the minimum, a processor thread will not be out into the pool again. Thus, the pool size always oscillates between max and min.

Setting `min` to a value greater than one means keeping a certain number of threads ready to service incoming requests without delay. This is especially useful if you now that requests are likely to come in in a bursty fashion. Limiting the pool size to a certain maximum is done to prevent servers from occupying all available ressources.

Request processor threads usually run at the highest thread priority. It is possible to influence thread priorities by setting the property `jacorb.poa.thread_priority` to a value between Java's Thread.MIN_PRIORITY and Thread.MAX_PRIORITY. If the confgured priority value is invalid JacORB will assign maximum priority to request processing threads.

Chapter 6

Implementation Repository

"... it is very easy to be blinded to the essential uselessness of them by the sense of achievement you get from getting it to work at all. In other words — and that is a rock-solid principle on which the whole of the Corporation's Galaxywide success is founded — their fundamental design flaws are completely hidden by their superficial design flaws."

D. Adams: So Long and Thanks for all the Fish

The Implementation Repository is not, as its name suggests, a database of implementations. Rather, it contains information about where requests to specific CORBA objects have to be redirected and how implementations can be transparently instantiated if, for a given request to an object, none is reachable. "Instantiating an implementation" means starting a server program that hosts the target object. In this chapter we give a brief overview and a short introduction on how to use the Implementation Repository. For more details please see [HV99].

6.1 Overview

Basically, the Implementation Repository (ImR) is an indirection for requests using persistent object references. A persistent object reference is one that was created by a POA with a PERSISTENT lifespan policy. This means that the lifetime of the object is longer than that of its creating POA. Using the Implementation Repository for objects the lifetime of which does not exceed the life time of its POA does not make sense as the main function of the Implementation Repository is to take care that such a process exists when requests are made — and to start one if necessary.

To fulfill this function, the ImR has to be involved in every request to "persistent objects". This is achieved by rewriting persistent object references to contain *not* the address of its server process but the address of the ImR. Thus, requests will initially reach the ImR and not the actual server — which may not exist at the time of the request. If such a request arrives at the ImR, it looks up the server information in its internal tables to determine if the target object is reachable or not. In the latter case, the ImR has to have information about how an appropriate server process can be started. After starting this server, the client receives a LOCATION_FORWARD exception from the ImR. This exception, which contains a new object reference to the actual server process now, is handled by its runtime system transparently. As a result, the client will automatically reissue its request using the new reference, now addressing the target directly.

6.2 Using the JacORB Implementation Repository

The JacORB Implementation Repository consists of two separate components: a repository process which need only exist once in a domain, and process startup deamons, which must be present on every host that is to start processes. Note that none of this machinery is necessary for processes that host objects with a TRANSIENT life time, such as used by the RootPOA.

First of all, the central repository process (which we will call ImR in the following) must be started:

```
$ imr [-n] [-p <port>] [-i <ior_file>][-f <file>][-b <file>]
```

The ImR is located using the configuration property `jacorb.ImplementationRepositoryURL`. This property must be set such that a http connection can be made and the ImR's IOR can be read. Next, startup daemons must be created on selected hosts. To do this, the following command must is issued on each host:

```
$ imr_ssd
```

When a startup daemon is created, it contacts the central ImR.

To register a program such that the ImR can start it, the following command is used (on any machine that can reach the ImR):

```
$ imr_mg add "AServerName" -l -c "jaco MyServer"
```

The `imr_mg` command is the generic way of telling the ImR to do something. It needs another command parameter, such as `add` in this case. To add a server to the ImR, an *implementation name* is needed. Here, it is `"AServerName"`. The `-l` switch tells the repository that the server is to be run on the local machine from which the command was issued. If another host is desired, use the `-h hostname` option. Finally, the ImR needs to know how to start the server. The string `"jaco MyServer"` tells it how. The format of this string is simply such that the server daemon can execute it (using the Java API call `exec()`), i.e. it must be intelligible to the target host's operating system. For a Windows machine, this could, e.g. be ”start jaco MyServer” to have the server run in its own terminal window, under Unix the same can be achieved with ”xterm -e jaco MyServer”.

For a client program to be able to issue requests, it needs an object reference. Up to this point, we haven't said anything about how persistent object references come into existence. Reference creation happens as usual, i.e. in the server application one of the respcetive operations on a POA is called. For a reference to be created as “persistent”, the POA must have been created with a PERSISTENT lifespan policy. This is done as in the following code snippet:

```
    /* init ORB and root POA */
    orb = org.omg.CORBA.ORB.init(args, props);
    org.omg.PortableServer.POA rootPOA =
        org.omg.PortableServer.POAHelper.narrow(
                        orb.resolve_initial_references("RootPOA"));
```

```
/* create policies  */

org.omg.CORBA.Policy [] policies = new org.omg.CORBA.Policy[2];
policies[0] = rootPOA.create_id_assignment_policy(
                            IdAssignmentPolicyValue.USER_ID);
policies[1] = rootPOA.create_lifespan_policy(
                            LifespanPolicyValue.PERSISTENT);

/* create POA */

POA myPOA = rootPOA.create_POA("XYZPOA",
                            rootPOA.the_POAManager(), policies);

/* activate POAs */
poa.the_POAManager().activate();
```

(Note that in general the id assignment policy will be `USER_ID` for a POA with persistent object references because this id will often be a key into a database where the object state is stored.) If a POA is created with this life span policy and the ORB policy "use_imr" is set, the ORB will try to notify the ImR about this fact so the ImR knows it need not start a new process for requests that target objects on this POA. To set the ORB policy, simply set the property `jacorb.use_imr=on`. The ORB uses another property, `jacorb.implname`, as a parameter for the notification, i.e. it tells the ImR that a process using this property's value as its *implementation name* is present. If the server is registered with the ImR, this property value has to match the implementation name that is used when registering.

The application can set these properties on the command line using `java -Djacorb.implname=MyName`, or in the code like this:

```
/* create and set properties */
java.util.Properties props = new java.util.Properties();
props.setProperty("jacorb.use_imr","on");
props.setProperty("jacorb.implname","StandardNS");

/* init ORB  */
orb = org.omg.CORBA.ORB.init(args, props);
```

There are a few things you have to consider especially when restoring object state at startup time or saving the state of your objects on shutdown. It is important that, at startup time, object initialization is complete when the object is activated because from this instant on operation calls may come in. The repository knows about the server when the first POA with a PERSISTENT lifespan policy registers, but does not forward object references to clients before the object is actually reachable. (Another, unreliable way to handle this problem is to increase the `jacorb.imr.object_activation_sleep` property, so the repository waits longer for the object to become ready again.)

When the server shuts down, it is equally important that object state is saved by the time the last POA in the server goes down because from this moment the Implementation Repository regards the server as down and will start a new one upon requests. Thus, a server implementor is responsible for avoiding reader/writer problems between servers trying to storing and restoring the. (One way of doing this is to use POA managers to set a POA to holding while saving state and to inactive when done.)

6.3 Server migration

The implementation repository offers another useful possibility: server migration. Imagine the following scenario: You have written your server with persistent POAs, but after a certain time your machine seems to be too slow to serve all those incoming requests. Migrating your server to a more powerful machine is the obvious solution. Using the implementation repository, client references do not contain addressing information for the slow machine, so server migration can be done transparently to client.

Assuming that you added your server to the repository, and it is running correctly.

```
$ imr_mg add AServerName -h  a_slow_machine  -l -c "jaco MyServer"
```

The first step is to *hold* the server, that means the repository delays all requests for that server until it is released again.

```
imr_mg hold AServerName
```

Now your server will not receive any requests for its registered POAs. If you can't shut your server down such that it sets itself down at the repository, i.e. your POAs are set to inactive prior to terminating the process, you can use

```
imr_mg  setdown AServerName
```

to do that. Otherwise your POAs can't be reactivated at the repository because they are still logged as active.

If you want your server to be restarted automatically, you have to tell the repository the new host and maybe a new startup command.

```
imr_mg edit AServerName -h the_fastest_available_machine -
c "jaco MyServer"
```

If your server can be restarted automatically, you now don't even have to start it manually, but it is instead restarted by the next incoming request. Otherwise start it manually on the desired machine now.

The last step is to release the server, i.e. let all delayed requests continue.

```
imr_mg release heinz
```

By now your server should be running on another machine, without the clients noticing.

Chapter 7

Interceptors

(Note: this chapter describes the proprietary JacORB interceptors only. As of now, there is no documentation of the Portable Interceptors other than the example in the `demo` directory. Please consider carefully whether you really need to use proprietary interceptors. Message–level interceptors will be deprecated soon!)

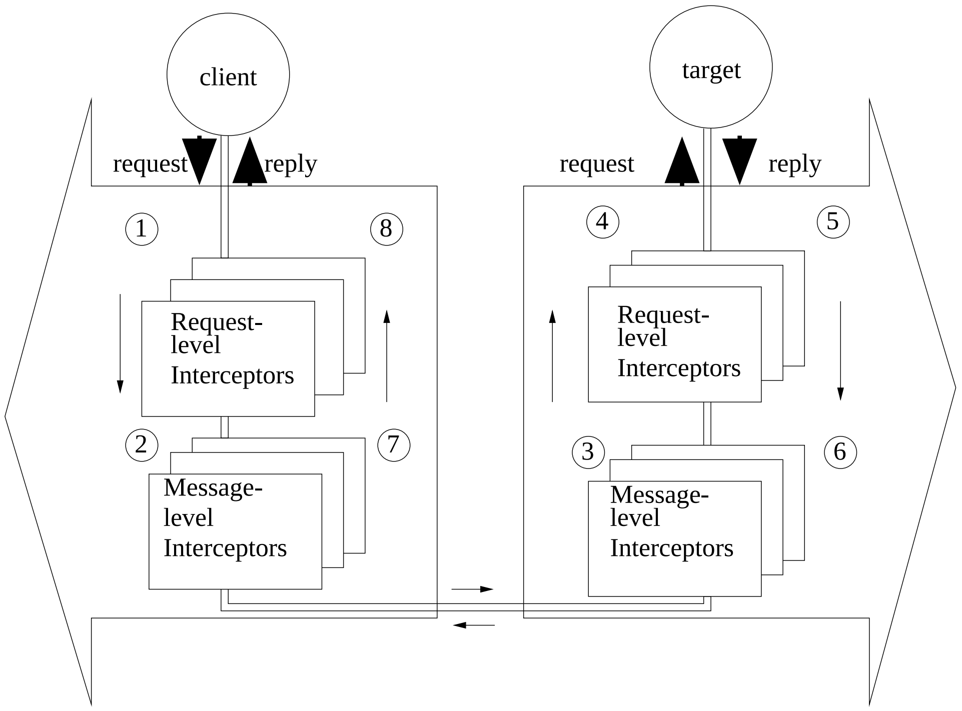
Interceptors are a flexible means for having code called by the ORB at specific points during operation invocations. Using interceptors you can, e.g., monitor and log method calls or define simple access control policies.

7.1 Introducing Interceptors

Interceptors are objects of particular classes which are called at specific points during the processing of operation invocations. By defining subclasses of interceptor classes and registering instances with the ORB, you can "insert" your own code into the invocation path. This provides access to the request itself and thus a convenient and very powerful way of observing and/or modifying requests. On the other hand, it requires more detailed knowledge about ORB data structures than necessary for application level code.

Interceptors can be defined at two levels: at the request level or at the message level. Request–level interceptors are given access to a request object representing the current request and may access and modify this request object before and after it is actually invoked. Message–level interceptors have access to the actual message buffer before and after it is sent across the network and can be used, e.g., to encrypt and decrypt the entire message.

In a CORBA invocation, there are eight points where interceptors are (potentially) given control. On the client side, these are points 1 and 2 (cf. figure), i.e. before the request is sent across the network, and points 7 and 8, i.e. after the result is received and before control is returned to the caller. At point 1, the interceptor has access to a structured request object, whereas at point 2, the message can be accessed in marshalled form in a buffer just before it is sent onto the network.


client
target
request
reply
request
reply
1
8
4
5
Request-level Interceptors
Request-level Interceptors
2
7
3
6
Message-level Interceptors
Message-level Interceptors

Because of this distinction, interceptors come in two flavours: request–level and message–level interceptors. On the server side, intercepting requests can also be done at these different levels (points 3, 4 and 5, 6). For every single one of these eight points, a different method is called on different interceptor objects. Let's first look at request–level interceptors.

Before we will explain the different kinds of interceptors in more details and look at some examples, please note that for interceptors to be usable in JacORB, they have to be enabled in the `.jacorb_properties` file. Per default, interceptors are disabled for performance reasons. If you want to enable them, you have to edit the following lines in `.jacorb_properties` to set the kind of interceptor you want to use to `on`:

```
# interceptor configuration
jacorb.interceptor.client.Requests=on
jacorb.interceptor.client.Messages=on
jacorb.interceptor.server.Messages=on
jacorb.interceptor.server.Requests=on
jacorb.interceptor.perObject=on
```

7.2 Request–level Interceptors

Request–level Interceptors are objects of subclasses of `ClientRequestInterceptor` on the client side and of `ServerRequestInterceptor` on the server side. We will look at these two in turn.

7.2.1 Client–side interceptors

On the client side, methods `pre_invoke()` and `post_invoke()` of classes implementing `jacorb.orb.ClientRequestInterceptor`[1] are called at points 1) and 8), respectively. Below is this interface:

```
package jacorb.orb;
public interface ClientRequestInterceptor
{
   public  void pre_invoke( org.omg.CORBA.Request r );
   public  void post_invoke( org.omg.CORBA.Request r );
}
```

Note that request information is available in the form of a *Dynamic Invocation Interface* (DII) request object at this level. Currently, JacORB does not automatically transform requests that were not issued using the DII into this form. Thus, request–level interceptors on the client side will only be called if the DII is used.

As an example, consider the following interceptor which traces method calls:

```
package jacorb.orb.util;

public class TraceClientRequestInterceptor
   implements jacorb.orb.ClientRequestInterceptor
{
   private int indent = 0;

   public TraceClientRequestInterceptor(){}

   public void pre_invoke( org.omg.CORBA.Request r )
   {
      for( int i = 0; i < indent; i++ )
         System.out.print(" ");
      System.out.println("[ invoke " + r.operation() + " ]");
      indent += 2;
   }
   public void post_invoke( org.omg.CORBA.Request r )
   {
      indent -= 2;
      for( int i = 0; i < indent; i++ )
         System.out.print(" ");
      System.out.println("[ " + r.operation() + " returns ]");
   }
}
```

[1]The OMG is in the process of defining interfaces for portable interceptors, so there is no standardized interface at present.

7.2.2 Server–side interceptors

Interceptors that are called at points 2) and 3) on the server side implement the interface `jacorb.orb.ServerRequestInterceptor`:

```
package jacorb.orb;

public interface ServerRequestInterceptor
{
   public  void pre_invoke( org.omg.CORBA.ServerRequest r );
   public  void post_invoke( org.omg.CORBA.ServerRequest r );
}
```

These interceptors gain access to the request through objects of type `org.omg.CORBA.ServerRequest` respectively. On the server side, JacORB does transform any request into a `org.omg.CORBA.ServerRequest`, so there is no restriction like the one pointed out above.

7.2.3 Per–object vs. per–process Interceptors

If the ORB is to use instances of the above class during invocation processing, these interceptors need to be registered with the ORB. There are two options for doing this. An interceptor can be registered for general use, i.e. it will be called during every single invocation from the current address space, regardless of which object is the target of the call. This is called a *per–process* interceptor.

Another, more fine–grained way of using interceptors is to register them as *per–object* interceptors, i.e. to register them for use in invocations on particular objects only. As an example, consider the simple tracer again: You may well want to trace method calls on all objects, or you might be interested in operations on one particular object only.

7.2.4 Registering Interceptors

A per–process interceptor is registered with the ORB as shown in the following code fragment:

```
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
    jacorb.orb.util.TraceClientRequestInterceptor t =
            new jacorb.orb.util.TraceClientRequestInterceptor();
    ...
    orb.addInterceptor( t );
    ...
```

When `addInterceptor()` is called, the ORB will append the argument as a per–process interceptor to the end of an internal interceptor chain. On any subsequent invocation, all interceptors in this chain will be called in the order in which they were appended to the chain. Of course, interceptors can also be removed again:

```
   orb.removeInterceptor(t);
```

Per–object interceptors are registered by calling `addInterceptor()` with the object reference as the first argument:

```
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
    jacorb.orb.util.TraceClientRequestInterceptor t =
            new jacorb.orb.util.TraceClientRequestInterceptor();
    org.omg.CORBA.Object o = ...
    ...
    orb.addInterceptor( o, t );
    ...
    orb.removeInterceptor( o, t);
```

7.3 Message–level Interceptors

Message–level Interceptors implement either `ClientMessageInterceptor` or `ServerMessageInterceptor` and generally work like request–level interceptors. The only difference is that the request is passed as an unstructured buffer.

Also note that the client–side interceptor has an additional parameter indicating the target of the invocation. This parameter is not available at the server–side because it is only known after the request message has been interpreted by the transport layer software — which is only after the message–level interceptor has had access to the message, e.g. for decryption purposes.

```
package jacorb.orb;
public interface ClientMessageInterceptor
{
   public  byte [] pre_invoke( org.omg.CORBA.Object t,
                                 byte [] buf
                              );

   public  byte [] post_invoke( org.omg.CORBA.Object t,
                                  byte [] buf
                              );
}

package jacorb.orb;
public interface ServerMessageInterceptor
{
   public  byte [] pre_invoke( byte [] buf );
   public  byte [] post_invoke( byte [] buf );
}
```

7.3.1 Example: A simple Encryption Interceptor

As an example, consider a simple interceptor which “encrypts” the message contents by reversing the message buffer. Note that the interceptor preserves the first 12 bytes of the message — this is the GIOP message header. The reason for this is that the message can only be correctly received if the receiving side is able to determine the message size. To this end, a GIOP message header with the correct message size *must* be sent across.

In the example, the transformation on the message buffer does not change the message size, so we can simply reuse the original message header. Many other transformations, however, *will* result in a different message length, so it is necessary to provide a correct message header. The format of the GIOP message header, as described in the CORBA specification, is as follows:

```
// IDL
struct Version
{
      char major;
      char minor;
};

struct MessageHeader
{
      char          magic [4];
      Version       GIOP_version;
      boolean       byte_order;
      octet           message_type;
      unsigned long message_size;
};
```

Alternatively, you can examine the class `org.omg.GIOP.MessageHeader` which gives the Java equivalent of the IDL specification. Here comes the Java code for the example “encryption” interceptor:

```
public class ClientMessageInterceptor
   implements jacorb.orb.ClientMessageInterceptor
{
   public  byte [] pre_invoke( org.omg.CORBA.Object t,
                               byte [] buf )
   {
      byte buf2[] = new byte[buf.length];

      // copy GIOP message header
      System.arraycopy( buf, 0, buf2, 0, 12 );

      // reverse rest of message
      for( int i = 12; i < buf.length; i++ )
           buf2[i] = buf[ (buf.length+11) - i ];

      return buf2;
   }

   public  byte [] post_invoke( org.omg.CORBA.Object t,
                                byte [] buf )
   {
      // don't do anything here
      return buf;
   }
}
```

On the receiving side, the effect of the buffer transformation must be undone before the message can be passed up to the ORB software and be interpreted as a request, so we must apply the same transformation again:

```
public class ServerMessageInterceptor
   implements jacorb.orb.ServerMessageInterceptor
{
   public  byte [] pre_invoke( byte [] buf )
   {
      byte buf2[] = new byte[buf.length];

      // copy GIOP message header
      System.arraycopy( buf, 0, buf2, 0, 12 );

      // reverse rest of message
      for( int i = 12; i < buf.length; i++ )
         buf2[i] = buf[ buf.length+11-i ];

      return buf2;
   }

   public  byte [] post_invoke( byte [] buf )
   {
   // don't do anything here
      return buf;
   }
}
```

7.3.2 Per–object vs. per–process Interceptors

As with request–level interceptors, it is sometimes useful to determine whether to apply a particular interceptor depending on which object is actually called. Accordingly, message–level interceptors can be registered per–object or per–process. However, this is only possible on the client side:

```
   orb.addInterceptor( (org.omg.CORBA.Object)s, new ClientMessageIn-
terceptor());
```

Recall that on the receiving side, a reference to the object is only available after the ORB software has interpreted the message — which it cannot do, e.g., before the message has been decrypted in the above example. As a consequence, server–side message–level interceptors can only be registered per process. If an interceptor may only apply certain transformations on messages to particular objects, it is responsible for determining the target by itself. The following code is from the server class in the interceptor example in `jacorb/demo/interceptors`:

```
    jacorb.orb.ORB orb = (jacorb.orb.ORB)org.omg.CORBA.ORB.init();
    org.omg.PortableServer.POA poa =
         org.omg.PortableServer.POAHelper.narrow(
               orb.resolve_initial_references("RootPOA"));

    poa.the_POAManager().activate();

    org.omg.CORBA.Object o = poa.servant_to_reference(new Server() );

    // create and register a trace interceptor
    //orb.addServerInterceptor( new TraceServerRequestIntercep-
tor() );

    jacorb.naming.NameServer.registerService( o, "Interceptors-
Example Server" );

    // important: let the interceptor registration be the last
    // thing in this server. Every subsequently received message
    // will be "decoded". This leads to chaos if there are messages
    // that are not "encoded", e.g. from the name server !

    orb.addServerInterceptor( new ServerMessageInterceptor() );
    orb.run();
```

Chapter 8

Applets — The JacORB Appligator

by Sebastian Müller

Since version 1.0 beta 13, JacORB includes an IIOP proxy called Appligator. Using this proxy, you can run Java applets with JacORB. Regular java programs can connect to every host on the internet — applets can only open connections to their applethost (the host they are downloaded from). This lets applets only use CORBA servers on their applethost, if no proxy is used. With JacORB Appligator, access for your applets is no longer restricted. Placed on the applethost, Appligator handles all connections from and to your applet transparently.

8.1 Using Appligator

Due to the transparency of JacORB Appligator you can write your applet as if it were a normal CORBA program. The only thing you have to do is to use a special initialization of the ORB: You have to call the `jacorb.orb.ORB(java.applet.Applet, java.util.Properties)` constructor.

A normal JacORB program reads a local `jacorb.properties` file to get the URL of its name server and other vital settings. An applet of course has no *local* properties file, but a remote one: You have to place the properties file (which has the same syntax and parameters as the normal file) in the same directory as your applet (the file name has to be: `jacorb.properties`, without a leading dot).

Similar to the name server, Appligator writes its IOR to a file. Your applet has to know the location of this file to retrieve the IOR of Appligator. You can set the location of the IOR file via the jacorb.properties file (parameter jacorb.ProxyServerURL) or with an applet parameter in the `<APPLET> HTML` tag (parameter `JacorbProxyServerURL`). If you give no (or a wrong) parameter JacORB will look for an IOR file called ”proxy.ior” in the codebase directory and in the web server root directory.

Make sure the parameter `jacorb.NameServerURL` points to a location on the applethost, otherwise you will get an applet security exception if your applet needs to use the name server.

Starting Appligator

Just type `appligator <filename>` to start the proxy. `<filename>` is the location where the IOR of the Appligator is written to. This location has to be specified in the jacorb.properties file of the applet or in an applet parameter (if it is not one of the standard paths, see below). If you want the appligator to start on a given port type `appligator -DOAPort=portnumber <filename>`

Summary

- init the ORB with `jacorb.orb.init(applet,properties)`, where applet is this applet and properties are `java.util.Properties` (which can be null)
- put a `jacorb.properties` file in the directory of the applet
- specify the location for the appligator IOR file in the `jacorb.properties` (jacorb.ProxyServerURL) or in an applet parameter (JacorbProxyServerURL)
- make sure the name server IOR file is accessible for the applet (lies on the applethost)
- start Appligator on the applethost (web server) with:
  `$ appligator <filename>`
  where filename is the location the appligator IOR is put to and has to be the location specified in the jacorb_properties or applet parameter.

Applet Properties

As described above there are some ways for the applet to get its jacorb properties. The most important property is the URL to the appligator IOR file. Without this property the applet will not work. If you use a name server, the URL to the name server IOR has to be specified too.

Properties can be set in three ways:

1. in the `ORB.init()` call with the `java.util.Properties` parameter
2. in the `jacorb_properties` file located in the same directory as the applet
3. the URL to the name server and Appligator IOR file can be set in the applet tag in the HTML file

Appligator and Netscape/IE, appletviewer

Netscape Navigator/Communicator comes with its own (outdated) CORBA support. You have to delete Netscape's CORBA classes to use JacORB. To do this you have to delete the file iiop10.jar located in `NS_ROOT/java/classes`. It's a good idea to store a backup of Netscape's file in an other directory. Then you can load the JacORB classes over the net together with you applet(put them in the codebase or specify a `ARCHIVE`).

Replacing the Netscape CORBA classes with JacORB's does not work (at least with Netscape Communicator 4.7), as Netscape does not allow local classes to listen on a socket.

If Netscape loads wrong classes or throws security exceptions (have a look at Netscape's JAVA Console to see this) be sure to check your `CLASSPATH`. Remove all JacORB and VisiBroker classes from your `CLASSPATH`.

Microsoft's Internet Explorer is stricter than Netscape: Even downloaded classes are not allowed to listen on a socket. Internet Explorer can only be used when the applet is a CORBA pure client which will not be called back.[1]

Appligator works well with Sun's appletviewer. You only have to make the appletviewer replace the Sun's CORBA classes with JacORB's classes. A typical appletviewer call for JacORB Applets looks like this (written in one command line):

```
appletviewer http://www.example.com/CORBA/dii_example.html
```

There is a shell script called "jacapplet" in JacORB's bin directory, which calls the appletviewer with the appropriate options (you have to edit it to match your local jdk path).

Examples

There are some example applets in the demo directory (`jacorb/demo/applet`). They are based on the normal examples. The examples include a HTML file which calls the applet. To run the example start the name server first. Start appligator on your web server and than the normal example server corresponding to the applet example on any computer in any order. Then you can call the example applet with the JDK appletviewer or Netscape.

Be sure to have a `jacorb.properties` file and the `jacorb.jar` in place.

[1] I believe that there must be a setting in Internet Explorer to allow incoming connections. Any ideas?

Chapter 9

IIOP over SSL

by André Benvenuti.

[Note: this is just “readme”–style information. A more detailed chapter needs to be written.]

9.1 Abstract

To have SSL support requires NO changes in your source code, but you need to rebuild JacORB as explained hereafter, create and deploy key stores holding the cryptographic data.

9.2 Required

- JDK 1.2 or later,
- IAIK-JCE 2.6 beta 1, the security provider classes downloadable from `http://jcewww.iaik.tu-graz.ac.at/`,
- iSaSiLk 3.0 or later, the SSL implementation from the same source.

[Remark: We will allow using SUN provider and SSL implementation in the next release]

9.3 Introduction

When the ORB instance is initialized by the application, properties are read from files and the command line. For the default SSL support we define two properties:

```
jacorb.security.support_ssl=on
jacorb.security.enforce_ssl=on
```

If `support_ssl=off` then `enforce_ssl=off` also. `enforce_ssl` means that any outgoing or incoming request will have to be over an SSL connection; `CORBA::NO_PERMISSION` is thrown if the policy is violated.

If `jacorb.security.support_ssl=on` the user will have to authenticate himself. The authentication succeeds if a key entry can be found in the key store file.

Cryptographic data (key pairs and certificates) is stored in a `.keystore` file. The `.keystore` has to be in the home directory (where property files also are). IMPORTANT: define in your property file

```
jacorb.security.keystore=.keystore
```

otherwise a .keystore file will be searched in the current directory.

The key store holds the key entries for principals, which can authenticate themselves to the object system, and the trusted certificate entries for the Certification Authorities accepted by the object system.

There is one instance of JacORB per JVM. This ORB instance will use the user's keys and certificate chain, when establishing SSL connections.

The class `jacorb.security.util.KeyStoreManager` should be used to create the key stores. Its features are: generate key pairs, sign public keys, import or export certificates chains and define trusted certificate authorities. The first certificate in the certificate chain may have a role extension.

9.4 Building JacORB with SSL support

1. Make sure that the cryptographic libraries are in your CLASSPATH. (If they are not, JacORB will be built without SSL support.)
2. Build JacORB anew: go to the installation directory and call ”ant”.
3. Edit the properties file to set the security-relevant properties as outlined above.

9.5 Creating key stores

You can proceed as follow:

1. Start jacorb.security.util.KeyStoreManager and create a (new) master key store, which should hold key pairs for the users and CAs (menu File/New).

   Remember that key pairs are stored in key entries. For each user and CA create a key entry (menu Keys/Create). Remark: a user can be assigned more then one key entry (use different aliases), if for example he will act in different roles. For more details, please consult the JKD tool documentation for ”keytool”, which explains the key store concept.
2. Public keys of users have to be signed by at least one CA (you can have intermediate signer in a certificate chain). As you create the certificate chain (menu Certificates/Create) you can assign a role to the alias.

3. Export the certificates of the CAs (menu Certificates/Export). Save your master key store to a file.

4. For each machine and/or user you will derive a key store as follows:

   (a) Make a copy of the master key store and open it in KeyStoreManager (menu File/Open).

   (b) Remove all key entries for CAs (menu Keys/Delete).

   (c) Remove all key entries for users not that machine (menu Keys/Delete).

   (d) Create a trusted certificate entry for each CA (menu Trustees/Add).You will need the exported certificates for this.

   (e) Save the key store to a file.

5. Deploy the key store files.

Chapter 10

JacORB utilities

In this chapter, we will briefly explain the executables that come with JacORB. These include the IDL–compiler, the JacORB name server.

10.1 idl

The IDL compiler parses IDL specifications and maps these to a set of Java classes. IDL interfaces are translated into Java interfaces, and typedefs, structs, const declarations etc. are mapped onto corresponding Java classes. Using `idl`, stubs and skeletons for all interface types in the IDL specification will automatically be generated.

Usage

```
idl [-h|-help] [-v|-version] [-syntax] [-all] [-Idir] [-
Dsymbol[=value]] [-d <Output Dir>] [-p <package_prefix>] [-i2jpackage
<mapping>] <filelist>
```

The option `-v` prints out a short version string while `-h` or `-help` displays brief usage information.

Invoking `idl` with the `-syntax` option allows you to check your IDL specification for syntactic errors without producing code. Without `-syntax`, the compiler creates directories according to the Java package structure.

The compiler does not, by default, generate code for included files. If that is desired, you have to use the `-all` option which causes code to be generated for every IDL file directly or indirectly included from within `<filelist>`. If you want to make sure that for a given IDL no code will be generated even if this option is set, you can use the (proprietary) preprocessor directive `#pragma inhibit_code_generation`.

The `-I` options allows you to specify one or more search paths for IDL files included from within `<filelist>`. If no path is given, only the current directory will be considered.

With the `-D` option you can define symbols that can be used by the preprocessor while processing the IDL file. If no value is specified, the symbol will be defined with a value of 1.

Compiling a file with a module `ModuleX` and an interface `InterfaceY` in it will result in a subdirectory `ModuleX` with `InterfaceY.java` in it (and possibly more `.java`–files). By default, the root directory for all of the files created during the process is the current directory. You can, however, provide a different directory in which these files are to be placed by using the -d option. Using the -d option when you are only checking the syntax of an IDL file has no effect.

With the `-p` option it is also possible to specify a package prefix for the generated Java classes. E.g., if the IDL file contains a module `Bank` which defines an interface `Account`, you can direct the IDL compiler to generate Java classes `example.Bank.Account` by compiling with the package prefix set to `example`, i.e. by compiling with

```
idl -p example bank.idl
```

The IDL compiler will create the appropriate directories if necessary. Note that the effect is the same as if the entire contents of the IDL file was scoped with `example`. In particular, this applies to included files: all definitions in included files will also be scoped this way! The -p switch should used with discretion in cases where other IDL files are included.

To be able to flexibly redirect generated Java classes into packages, the `-i2jpackage` switch can be used. Using this option, any IDL scope x can be replaced by one (or more) Java packages y. Specifying `-i2jpackage X:a.b.c` will thus cause code generated for IDL definitions within a scope x to end up in a Java package `a.b.c`, e.g. an IDL identifier `X::Y::ident` will be mapped to `a.b.c.y.ident` in Java.

(The IDL parser was generated with Scott Hudson's CUP parser generator. The LALR grammar for the CORBA IDL is in the file `jacorb/Idl/parser.cup`.)

10.2 ns

JacORB provides a service for mapping names to network references. The name server itself is written in Java like the rest of the package and is a straightforward implementation of the CORBA "Naming Service" from Common Object Services Spec., Vol.1 [OMG97]. The IDL interfaces are mapped to Java according to our Java mapping.

Usage

```
ns <filename> <filename> [<timeout>]
```

or

```
jaco jacorb.Naming.NameServer <filename> <filename> [<timeout>]
```

Example

```
ns ˜/public_html/NS_Ref ˜/nsdb
```

The name server does *not* use a well known port for its service. Since clients cannot (and need not) know in advance where the name service will be provided, we use a bootstrap file in which the name server records an object reference to itself (its *Interoperable Object Reference* or IOR). The name of this bootstrap file has to be given as an argument to the `ns` command. This bootstrap file has to be available to clients network-wide, so we demand that it be reachable via a URL — that is, there must be an appropriately configured HTTP server in your network domain which allows read access to the bootstrap file over a HTTP connection. (This implies that the file must have its read permissions set appropriately. If the binding to the name service fails, please check that this is the case.) After locating the name service through this mechanism, clients will connect to the name server directly, so the only HTTP overhead is in the first lookup of the server.

The name bindings in the server's database are stored in and retrieved from a file that is found using the second file name argument. When the server starts up, it tries to read this file's contents. If the file is empty or corrupt, it will be ignored (but overriden on exit). The name server can only save its state when it goes down after a specified timeout. If the server is interrupted (with `CTRL-C`), state information is lost and the file will not contain any usable data.

If no timeout is specified, the name server will simply stay up until it is killed. Timeouts are specifed in milliseconds.

10.3 nmg

The JacORB NameManager, a GUI for the name service, can be started using the `nmg` command. The NameManager then tries to connect to an existing name service.

Usage

```
nmg
```

10.4 lsns

This utility lists the contents of the default naming context. Only currently active servers that have registered are listed. The `-r` option recursively lists the contents of naming contexts contained in the root context. If the graph of naming contexts contains cycles, trying to list the entire contents recursively will not return...

Usage

```
lsns [ -r ]
```

Example

```
$ lsns
/grid.service
```

when only the server for the grid example is running and registered with the name server.

10.5 dior

JacORB comes with a simple utility to decode an interoperable object reference (IOR) in string form into a more readable representation.

Usage

```
dior <IOR-string> | -f <filename>
```

Example

In the following example we use it to print out the contents of the IOR that the JacORB name server writes to its file:

```
dior -f ~/public_html/NS_Ref
------IOR components-----
TypeId  :       IDL:omg.org/CosNaming/NamingContextExt:1.0
Profile Id   :  TAG_INTERNET_IOP
IIOP Version :  1.0
Host    :       160.45.110.41
Port    :       49435
Object key :    0x52 6F 6F 74 50 4F 41 3A 3A 30 D7 D1 91 E1 70 95 04
```

10.6 pingo

"Ping" an object using its stringified IOR. Pingo will call `_non_existent()` on the object's reference to determine whether the object is alive or not.

Usage

```
pingo <IOR-string> | -f <filename>
```

Chapter 11

Limitations, Feedback

A few limitations and known bugs (list is incomplete):

- the IDL compiler does not support
  - the `context` construct
- the API documentation and this document are incomplete.

Feedback and bug reports

Please mail bug reports as well as criticism and experience reports to:

`brose@inf.fu-berlin.de`

Bibliography

[Bro98] Gerald Brose. Reflection in Java, CORBA and JacORB. In C. Cap, editor, *Proc. Java–Informationstage (JIT)*, pages 238–248. Springer, 1998.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison–Wesley, 1999.

[OMG97] OMG. *CORBAservices: Common Object Services Specification*, November 1997.

[OMG98] OMG. *Java to IDL Mapping, Joint Submission*, 1998.

[Sie96] Jon Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996.

[VD98] Andreas Vogel and Keith Duddy. *Java Programming with CORBA*. John Wiley and Sons, 2 edition, 1998.

[Vin97] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.

[Vin98] Steve Vinoski. New features for corba 3.0. *CACM*, 41(10):44–52, October 1998.