JacORB 1.3 Programming Guide

Gerald Brose, Nicolas Noffke, Sebastian Müller
Institut für Informatik
Freie Universität Berlin, Germany
{brose,noffke,semu}@inf.fu-berlin.de

$Revision : 1.5$

May 2, 2001

Contents

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, a couple of CORBA Object Service implementations, and a number of example programs. This document is *not* an introduction to CORBA in general. Please see [BVD01, Sie00, Vin97] for this purpose and [HV99] for more advanced issues. The JacORB version described in this document is JacORB 1.3. 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.

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://jacorb.inf.fu-berlin.de/`. It can also be downloaded via anonymous ftp from `ftp.inf.fu-berlin.de` from the directory `pub/jacorb/`.

To install JacORB, just unzip and untar (or simply unzip) the archive somewhere. This will result in a new directory `JacORB1_3`. Make sure your CLASSPATH environment variable contains `JacORB1_3/lib/jacorb.jar`. If you plan to recompile all or parts of JacORB, you should also include `JacORB1_2/classes`. Extend your search path with `JacORB1_3/bin`, so that the shell scripts and batch files for the utilities in this directory are found.

2.2 Installation

2.2.1 Ant and build 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 NameManager, ImRManager, IRBrowser, 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 the cryptography and SSL libraries by IAIK, viz. IAIK–JCE 2.5 and iSaSiLk 3.0. Please see `http://jcewww.iaik.tu-graz.ac.at/`.

2.2.2 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 modules. There 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. If it finds multiple files, it loads all of them 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.Properties` 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"); // use put() un-
der Java 1.1
    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. This option has the disadvantage that you cannot use `jaco` to start the Java interpreter, which means that you must pass in the properties that this script sets.

```
$ java -DOAPort=4711 -D...  -D...  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
##

########################################
#                                      #
#   Initial references configuration   #
#                                      #
########################################

#
# URLs where IORs are stored (used in orb.resolve_initial_service())
# DO EDIT these! (Only those that you are planning to use,
```

```
# of course ;-).
#
# The ORBInitRef references are created on ORB startup time. In the
# cases of the services themselves, this may lead to exceptions being
# displayed (because the services aren't up yet). These exceptions
# are handled properly and cause no harm!

#ORBInitRef.NameService=corbaloc::160.45.110.41:38693/\
#                       StandardNS/NameServer%2DPOA/_root
#ORBInitRef.NameService=file:/c:/NS_Ref
ORBInitRef.NameService=http://www.x.y.z/˜user/NS_Ref
#ORBInitRef.TradingService=http://www.x.y.z/˜user/TraderRef

# JacORB-specific URLs
jacorb.ImplementationRepositoryURL=http://www.x.y.z/˜user/ImR_Ref
jacorb.ProxyServerURL=http://www.x.y.z/˜user/Appligator_Ref

##################################
#                                #
#   Debug output configuration   #
#                                #
##################################

# use (java) jacorb.util.CAD to generate an appropriate
# verbosity level
# 0 = off
# 1 = important messages and exceptions
# 2 = informational messages and exceptions
# >= 3 = debug-level output (may confuse the unaware user :-)
jacorb.verbosity=1

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

##################################################
#                                                #
#    WARNING: The following properties should    #
#    only be edited by the expert user. They     #
#    can be left untouched for most cases!       #
#                                                #
##################################################

################################
#                              #
#   Basic ORB Configuration    #
#                              #
################################

# number of retries if connection cannot directly be established
jacorb.retries=5
```

```
# how many msecs. do we wait between retries
jacorb.retry_interval=500

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

# client-side timeout, set no non-zero to stop blocking
# after so many msecs.
#jacorb.connection.client_timeout=0

# max time a server keeps a connection open if nothing happens
#jacorb.connection.server_timeout=10000

#jacorb.reference_caching=off
...

#########################
#                       #
#   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=5

# if set, request processing threads in thePOA
# will run at this priority. If not set or invalid,
# MAX_PRIORITY will be used.
#jacorb.poa.thread_priority=

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

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 value for `ORBInitRef.NameService` is a URL for a resource used to set up the JacORB name server. This URL will be used by the ORB to locate the file used to store the name server's object reference (see also chapter 4).

The `verbosity` option tells JacORB how much diagnostic output it should emit at run-time. Unless the `logfile` property is set to a file name, diagnostic output will 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. If you like to selectively suppress specific output you can use the tool `jacorb.util.CAD` to generate a special verbosity level.

The `jacorb.poa.monitoring` property determines whether the POA should bring up a monitoring GUI for servers that let you examine the dynamic behavior 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 chapter 5 on the POA for more details.

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

3 Getting Started

Before we explain 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 build file 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_3/demo/grid`.

```
// server.idl
// IDL definition of a 2-D grid:
module demo
{
  module grid
  {
    interface MyServer
    {
        typedef fixed <5,2> fixedT;

        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 fixedT value);

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

        exception MyException
        {
            string why;
        };

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

3.3 Generating Java classes

Feeding this file into the IDL compiler

```
$ idl -d ../..  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 explained in detail in [BVD01]. 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. 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. It uses the Java library class `java.math.BigDecimal` for values of the IDL fixed–point type `fixedT`:

```
package demo.grid;

/**
 * A very simple implementation of a 2-D grid
 */

import demo.grid.MyServerPackage.MyException;

public class gridImpl
    extends MyServerPOA
{
    protected short height = 31;
    protected short width = 14;
    protected java.math.BigDecimal[][] mygrid;

    public gridImpl()
    {
        mygrid = new java.math.BigDecimal[height][width];
        for( short h = 0; h < height; h++ )
```

```
        {
            for( short w = 0; w < width; w++ )
            {
                mygrid[h][w] = new java.math.BigDecimal("0.21");
            }
        }
    }

    public java.math.BigDecimal get(short n, short m)
    {
        if( ( n <= height ) && ( m <= width ) )
            return mygrid[n][m];
        else
            return new java.math.BigDecimal("0.01");
    }

    public short height()
    {
        return height;
    }

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

    public short width()
    {
        return width;
    }

    public short opWithException()
        throws demo.grid.MyServerPackage.MyException
    {
        throw new demo.grid.MyServerPackage.MyException("This is only a test ex-
ception, 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")));
             }

             short x = grid.height();
             System.out.println("Height = " + x);

             short y = grid.width();
             System.out.println("Width = " + y);

             x -= 1;
             y -= 1;

             System.out.println("Old value at (" + x + "," + y +"): " +
                                grid.get( x,y));

             System.out.println("Setting (" + x + "," + y +") to 470.11");

             grid.set( x, y, new java.math.BigDecimal("470.11"));

             System.out.println("New value at (" + x + "," + y +"): " +
                                grid.get( x,y));

            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 /home/me/public_html/NS_Ref
```

where `/home/me/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 name server 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
Old value at (30,13): 0.21
Setting (30,13) to 470.11
New value at (30,13): 470.11
MyException, reason: This is only a test exception, no harm done :-)
done.
```

3.6.1 The Tie Approach

If your implementation class cannot inherit from the generated servant class `MyServerPOA` because, e.g., you need to inherit from another base class, you can use the tie approach. Put simply, it replaces inheritance by delegation. Instead of inheriting from the generated base class, your implementation needs to implement the generated *operations interface* `MyServerOperations`:

```
package demo.grid;
```

```
import demo.grid.MyServerPackage.MyException;

public class gridOperationsImpl
    implements MyServerOperations
{
...
}
```

Your server is then written as follows:

```
package demo.grid;

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

public class TieServer
{
    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"));

            // use the operations implementation and wrap it in
            // a tie object

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

            poa.the_POAManager().activate();

            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
            {
                NamingContextExt nc =
                     NamingContextExtHelper.narrow(
                        orb.resolve_initial_references("NameService"));
                NameComponent [] name = new NameComponent[1];
                name[0] = new NameComponent("grid", "whatever");
                nc.bind( name, o );
            }
        }
        catch ( Exception e )
        {
            e.printStackTrace();
        }
        orb.run();
    }
}
```

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 supports binding names to object references and to lookup object references using these names. It also allows clients 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 (and above) with the JacORB naming service: JDK 1.2 comes with a couple of outdated and apparently buggy naming service classes that do 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 in your code. 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>]
```

You can also start the Java interpreter explicitly by typing

```
$ jaco jacorb.naming.NameServer <filename> [<timeout>]
```

In the example

```
$ ns /home/me/public_html/NS_Ref
```

we direct the name server process to write location information and logging information to the file `/home/me/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 `ORBInitRef.NameService` 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];
name[0] = new NameComponent("sub","context");
NamingContextExt subContext =
  NamingContextExtHelper.narrow( rootContext.bind_new_context( name ));
```

Please note that the JacORB naming service always uses `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. Figure 4.1 gives a screen shot.

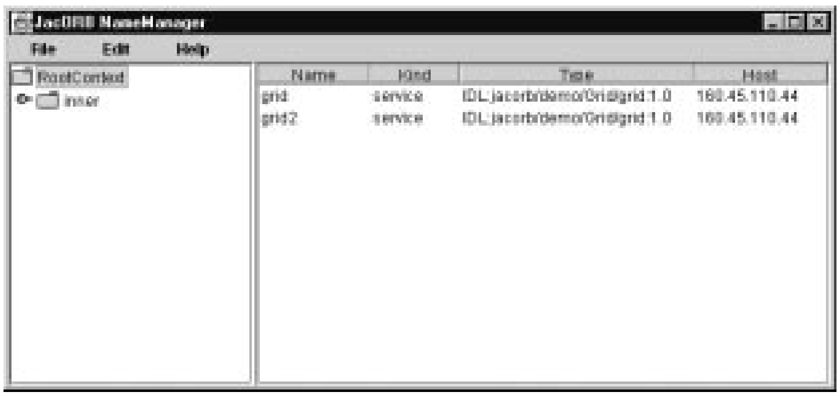


Figure 4.1: NameManager Screenshot

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.

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, the Portable Object Adapter (POA), and threading.

This chapter gives only a very superficial introduction to the POA. A thorough explanation of how the POA can be used in different settings and of the different policies and strategies it offers is beyond our scope here, but can be found in [BVD01]. Other references that explain the POA are [HV99, Vin98]. More in–depth treatment 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`. The ultimate reference, of course, is the CORBA specification.

5.1 POA

The POA provides a comprehensive set of interfaces for managing object references and servants. The code written using the POA interfaces is now portable across ORB implementations and has the same semantics in every ORB that is compliant to CORBA 2.2 or above.

The POA defines standard interfaces to do the following:

- Map an object reference to a servant that implements that object
- Allow transparent activation of objects
- Associate policy information with objects
- Make a CORBA object persistent over several server process lifetimes

In the POA specification, the use of pseudo-IDL has been deprecated in favor of an approach that uses ordinary IDL, which is mapped into programming languages using the standard language mappings, but which is locality constrained. This means that references to objects of these types may not be passed outside of a server's address space. The POA interface itself is one example of a locality–constrained interface.

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 currently 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 `jacorb.poa.thread_pool_max`. Otherwise, request processing is blocked until a thread is returned to the pool. Upon returning threads that have 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 resources.

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 configured priority value is invalid JacORB will assign maximum priority to request processing threads.

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 daemons, 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>] [-a]
```

The ImR is located using the configuration property `ORBInitRef.ImplementationRepository`. 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" -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"`. If the host were the server should be restarted is not the local one, 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"`.

The startup command is a string that is passed as the *single* argument to javas `Runtime.exec()` method, without interpreting it or adding anything. Since `Runtime.exec()` has system–dependent behaviour, the startup string has to reflect that. While for most unix systems it is sufficient to avoid shell–expansions like * and ˜, windows–based systems do not pass the string to a commandline interpreter so a simple `jaco MyServer` will fail even if it works if directly typed in at the dos prompt. Therefore you have to "wrap" the core startup command in a call to a commandline interpreter. On NT the following startup command will do the job: `cmd /c "jaco MyServer"`. Please keep in mind that if you use the `imr_mg` command to set the startup command, you have to escape the quotes so they appear inside of the resulting string.

If you don't intend to have your server automatically started by the ImR you can also set the property "`jacorb.imr.allow_auto_register`" or use the `-a` switch of the ImR process. If this property is set, the ImR will automatically create a new entry for a server on POA activation, if the server has not been registered previously. In this case you don't have to use the ImR Manager to register your server.

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 respective 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 lifespan policy and the ORB property “use_imr” is set, the ORB will try to notify the ImR about this fact so the ImR knows it doesn’t need to 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","MyName");

/* 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 store and restore the object state. (One way of doing this is to use POA managers to set a POA to holding while saving state and to inactive when done.)

Please keep in mind that even if you don't have to save the state of your objects on server shutdown you *must* deactivate your POAs prior to exiting your process (or at least use `orb.shutdown(...)` which includes POA deactivation). Otherwise the ImR keeps the server as active and will return invalid IORs. In case of a server crash you can use the command `imr_mg setdown AServerName` to notify the ImR of the server termination.

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 -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 AServerName
```

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

6.4 A Note About Security

Using the imr can pose a major security threat to your system. Imagine the following standard setup: an imr is running on a machine, its IOR file is placed in a directory where it can be read by the web server, and several imr_ssds are running on other machines. An attacker can now execute processes on the machines the ssds are running on by taking the following steps:

1. Setting the `ORBInitRef.ImplementationRepository` property to the IOR file on your server.
2. Creating a new logical server with the desired command to execute as startup command on the desired host (where a ssd is running). This is the crucial point. The ssd calls `Runtime.exec()` with the supplied string, and there is no way to check if the command does what it is supposed to do, i.e. start a server.
3. Start the server with the imr_mg. The startup command of the server will b exec'd on the specified host.

Now this should not generally discourage you to use the imr but show you that there are risks, which can be reduced significantly nonetheless. There are several ways to encounter this threat and we don't consider this list to be complete:

1. Try to control the distribution of the IOR file. Hiding it should not be considered here, because *security by obscurity* is generally a bad approach. Try to make use of file system mechanisms like groups and ACLs.
2. Use a firewall which blocks of incoming traffic. Keep in mind that if the attacker is inside of your protection domain, the firewall won't help. It is also not that hard to write a Trojan that can tunnel those firewalls that block incoming traffic.
3. Enforce SSL connections to the imr. This blocks all client connections that don't have a certificate signed by a CA of your choice. See chapter 9 for more information.

7 Interface Repository

Run–time type information in CORBA is managed by the ORB's *Interface Repository* (IR) component. It allows to request, inspect and modify IDL type information dynamically, e.g., to find out which operations an object supports. Some ORBs may also need the IR to find out whether a given object's type is a subtype of another, but most ORBs can do without the IR by encoding this kind of type information in the helper classes generated by the IDL compiler.

In essence, the IR is just another remotely accessible CORBA object that offers operations to retrieve (and in theory also modify) type information. Note that the JacORB IR is only available if you are using JDK 1.2 or above.

7.1 Type Information in the IR

The IR manages type information in a hierarchical containment structure that corresponds to the structure of scoping constructs in IDL specifications: modules contain definitions of interfaces, structures, constants etc. Interfaces in turn contain definitions of exceptions, operations, attributes and constants. Figure 7.1 illustrates this hierarchy.

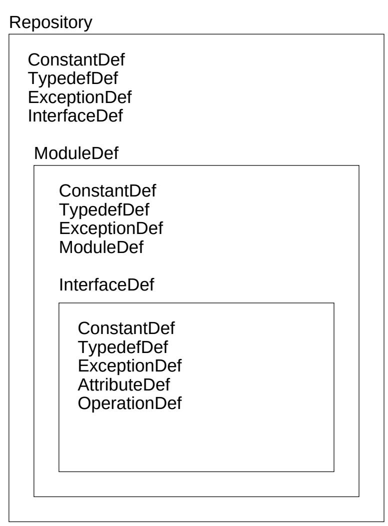


Figure 7.1: Containers in the Interface Repository

The descriptions inside the IR can be identified in different ways. Every element of the repository has a unique, qualified name which corresponds to the structure of name scopes in the IDL specification. An interface `I1` which was declared inside module `M2` which in turn was declared inside module `M1` thus has a qualified name `M1::M2::I1`. The IR also provides another, much more flexible way of naming IDL constructs using *Repository Id*s. There are a number of different formats for RepositoryIds but every Repository must be able to handle the following format, which is marked by the prefix `"IDL:"` and also carries a suffix with a version number, as in, e.g., ”`IDL:jacorb/demo/grid:1.0`”. The name component between the colons can be set freely using the IDL compiler directives `#pragma prefix` and `#pragma ID`. If no such directive is used, it corresponds to the qualified name as above.

7.2 Repository Design

When designing the Interface Repository, our goal was to exploit the Java reflection API’s functionality to avoid having to implement an additional data base for IDL type descriptions. An alternative design is to use the IR as a back-end to the IDL compiler, but we did not want to introduce such a dependency and preferred to a have a rather “light–weight” repository server. As it turned out, this design was possible because the similarities between the Java and CORBA object models allow us to derive the required IDL information at run time. As a consequence, we can even do without any IDL at compile time. In addition to this simplification, the main advantage of our approach lies in avoiding redundant data and possible inconsistencies between persistent IDL descriptions and their Java representations, because Java classes have to be generated and stored anyway.

Thus, the Repository has to load Java classes, interpret them using reflection and translate them into the appropriate IDL meta information. To this end, the repository realizes a reverse mapping from Java to IDL. Figure 7.2 illustrates this functionality, where f^{-1} denotes the reverse mapping, or the inverse of the language mapping.

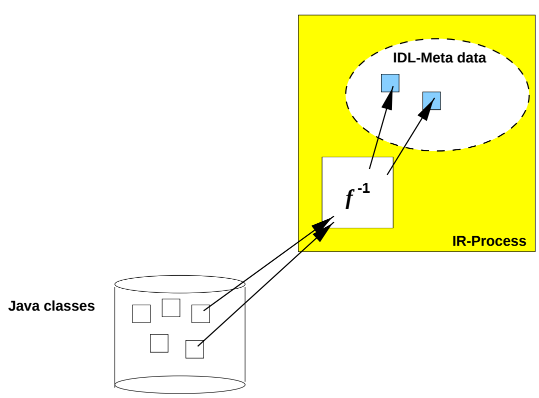


Figure 7.2: The JacORB Interface Repository

7.3 Using the IR

For the ORB to be able to contact the IR, the IR server process must be running. To start it, simply type the `ir` command and provide the required arguments:

```
$ ir /home/brose/classes /home/brose/public_html/IR_Ref
```

The first argument is a path to a directory containing `.class` files and packages. The IR loads these classes and tries to interpret them as IDL compiler–generated classes. If it succeeds, it creates internal representations of the adequate IDL constructs. The second argument on the command line above is simply the name of the file where the IR stores its object reference for ORB bootstrapping.

To view the contents of the repository, you can use the GUI IRBrowser tool or the query command. First, let's query the IR for a particular repository ID. JacORB provides the command `qir` ("query IR") for this purpose:

```
$ qir IDL:raccoon/test/cyberchair/Paper:1.0
```

As result, the IR returns an InterfaceDef object, and `qir` parses this and prints out:

```
interface Paper
{
   void read(out string arg_0);
   raccoon::test::cyberchair::Review getReview(in long arg_0);
   raccoon::test::cyberchair::Review submitReview(
       in string arg_0, in long a rg_1);
   void listReviews(out string arg_0);
};
```

To start the IRBrowser, simply type

```
$ irbrowser
```

Figure 7.3 gives a screen shot of the IR browser.

The Java classes generated by the IDL compiler using the standard OMG IDL/Java language mapping do not contain enough information to rebuild all of the information contained in the original IDL file. For example, determining whether an attribute in an interface was `readonly` or not is not possible, or telling the difference between `in` and `inout` parameter passing modes. Moreover, IDL modules are not explicitly represented in Java, so telling whether a directory in the class path represents an IDL module is not easily possible. For these reasons, the JacORB IDL compiler generates a few additional classes that hold the required extra information if the compiler switch `-ir` is used when compiling IDL files:

```
$ idl -ir myIdlFile.idl
```

The additional files generated by the compiler are:

- a `_XModule.java` class file for any IDL module X
- a `YIRHelper.java` class file for any interface Y.

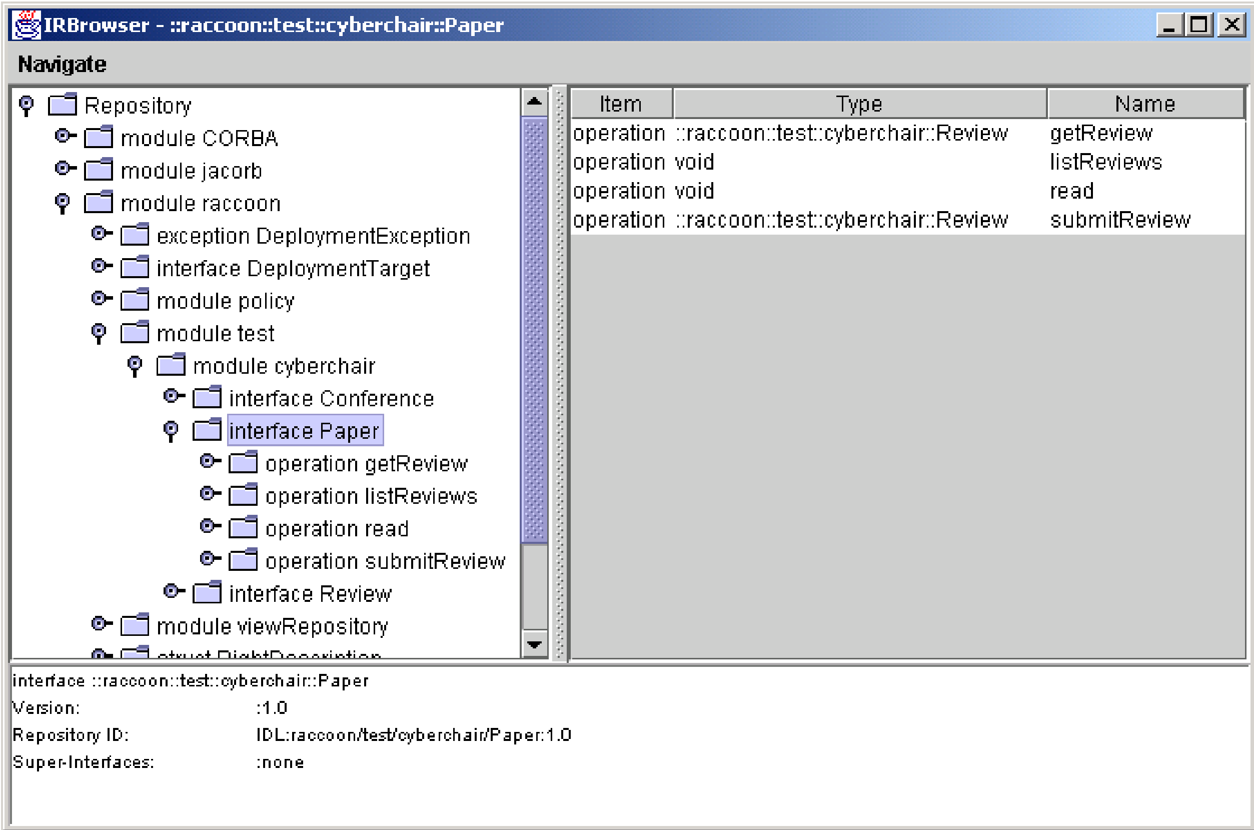


Figure 7.3: IRBrowser Screenshot

If no `.class` files that are compiled from these extra classes are found in the class path passed to the IR server process, the IR will not be able to derive any representations. Note that the IDL compiler does not make any non–compliant modifications to any of the standard files that are defined in the Java language mapping — there is only additional information.

One more caveat about these extra classes: The compiler generates the `_XModule.java` class only for genuine modules. Java package scopes created by applying the `-d` switch to the IDL compiler do not represent proper modules and thus do not generate this class. Thus, the contents of these directories will not be considered by the IR.

When an object's client calls the `get_interface()` operation, the ORB consults the IR and returns an `InterfaceDef` object that describes the object's interface. Using `InterfaceDef` operations on this description object, further description objects can be obtained, such as descriptions for operations or attributes of the interface under consideration.

The IR can also be called like any other CORBA object and provides `lookup()` or `lookup_name()` operations to clients so that definitions can be searched for, given a qualified name. Moreover, the complete contents of individual containers (modules or interfaces) can be listed.

Interface Repository meta objects provide further description operations. For a given `InterfaceDef` object, we can inspect the different meta objects contained in this object (e.g., `Opera-`

`tionDef` objects). It is also possible to obtain descriptions in form of a simple structure of type `InterfaceDescription` or `FullInterfaceDescription`. Since structures are passed by value and a `FullInterfaceDescription` fully provides all contained descriptions, no further —possibly remote — invocations are necessary for searching the structure.

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 <port> <filename>
```

to start the proxy on port `port`. `<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).

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 another directory. Note that renaming this jar file in the original directory does not suffice if you don't also change the `.jar` extension because Netscape loads all jar files in this directory. You then need to install jacorb.jar in this directory.

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` and look for old or . Remove all JacORB and

VisiBroker classes from your CLASSPATH. We succeeded running JacORB applet clients on Netscape 4.72 with the Java 1.3 plugin.

Microsoft's Internet Explorer is stricter than Netscape: Even downloaded classes are not allowed to listen on a socket. We strongly advise to use Sun's Java 1.3 plugin with IE also. To trick IE into using JacORB, you need to copy JacORB classes to `$WINNT`
`Java`
`TrustLib`. You can either copy the entire jacorb.jar and unpack it in this directory or just copy the directories `jacorb`, `org`, and `HTTPClient`.

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).

If you use the Appligator with other browsers or if you know a way to load the JacORB classes without removing and copying jars please let us know.

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.

8.2 Using JacORB with Firewalls

Typically firewalls do two things: filter traffic by port, and filter traffic by protocol. JacORB comes with two utilities to overcome firewall filtering. The JacORB Appligator can be used to deal with port restrictions, and HTTP tunneling helps you to tunnel through firewalls that are blocking GIOP messages.

8.2.1 The JacORB Appligator

The Appligator was written to avoid the sandbox restrictions for Java applets. Unsigned applets can only have connections to the host they are loaded from, which makes them useless in most distributed CORBA scenarios. The Appligator is a GIOP proxy, which enables applets to connect to every CORBA server by redirecting the traffic from the applet to the CORBA server to the proxy. The Appligator also works the other way round: Every connection the applet is redirected to the Appligator.

Even without applets the Appligator can be used as a GIOP proxy on a firewall. The Appligator is a CORBA object itself and is explicitly started on a given port using a command–line argument `<portnumber>`. All incoming traffic to the Appligator will go to port `<portnumber>`. If you configure your CORBA object behind the firewall to be aware of the Appligator all traffic from and to this objects will go through the Appligator (a better way would be to distinguish between traffic going over the firewall and traffic within the enclave, this has to be implemented in future versions of the Appligator).

To make your port filtering firewall working with CORBA and GIOP messages you must ask your system administrator to assign a port for GIOP messages on the firewall. Start the Appligator on this port.

Now all CORBA servers (which are aware of the Appligator) in your enclave can be contacted over the Appligator. If your CORBA client wants to contact a server in the Internet outside the firewall the connection will go over the Appligator. Callbacks from the Internet to your client do not work with Netscape.

To make your non–Applet clients and your CORBA objects aware of the Appligator you have to add a property in your jacorb.property file. Applets will use the Appligator automatically. If you want your applications to use the Appligator (which is true if you want to use the Appligator as a firewall proxy) than set `jacorb.use_appligator_for_applications = on`. If you want to turn off appligator use set `jacorb.use_appligator_for_applications = off` for applications and `jacorb.use_appligator_for_applets = off` for applets.

Finally you have to specify the location on the Appligator. This is done the same way as JacORB determines to location of the name server: When the Appligator starts the IOR of the Appligator is written to a file which is put to the location you specified as command parameter. This file must be accessible to all clients that want to use the Appligator. You can use a shared file system to access this file or put it on a web server etc. The location of the file in which the IOR of the Appligator is stored must be set in the jacorb.properties file. Use the ”jacorb.ProxyServerURL” property for this.

8.2.2 HTTP tunneling

If your firewall filters traffic by protocol and is not configured to allow GIOP messages you can use HTTP tunneling to make your GIOP traffic look like HTTP traffic to the firewall. HTTP tunneling is a built–in feature of JacORB and produces correct HTTP 1.1 messages so that your firewalls sees a HTTP request (from your CORBA client to the CORBA server) and a HTTP reply (from the CORBA server to your client) for each method you call on the server.

HTTP tunneling in JacORB is not compatible with any OMG standard so it will work between JacORB clients and servers only. Every JacORB server will recognize incoming HTTP traffic and will handle it correctly without any special settings needed. Configuration has to be done on the client side. You can configure HTTP tunneling on a IP address basis. If you set HTTP tunneling on for a given IP address, all traffic will be tunneled to all CORBA objects on this host. You can specify more than one IP address by using a comma separated list. Example:If you set `jacorb.use_httptunneling_for = 192.168.0.1, 192.168.0.2` all traffic to 192.168.0.1 and 192.168.0.2 will be tunneled.

8.2.3 Appligator and tunneling

As the Appligator is a normal JacORB CORBA object it supports tunneling, too. No extra configuration is needed to use the Appligator on a firewall and tunnel the firewall with HTTP.

8.2.4 Summary

- use the Appligator as a GIOP proxy an the firewall if your firewall is configured to block all traffic but traffic on some special ports.
- ask your system administrator to assign a special port for GIOP on your firewall and start the Appligator on this port on the firewall: for example

  `$ appligator 7777 Appligator_Ref`
- all CORBA objects that should be reachable from outside the firewall or need to contact a CORBA object outside the firewall must use the Appligator as a proxy. Add the `jacorb.use_appligator_for_applications = on` property to the jacorb.properties file for those applications
- Set the location of the Appligator in the jacorb.properties file of your clients (`jacorb.ProxyServerURL`)
- If you need HTTP tunneling because a firewall between your host and server is filtering GIOP messages use the property `jacorb.use_httptunneling_for` with the IP address of the server you want to use tunneling to

Final remarks

I want to thank Wendell Duncan for testing and debugging the tunneling code. I do not have a real firewall here in university, so I hope all the stuff works in a real environment :-). If you have problems feel free to contact me. I would be glad to hear if you successfully use the Appligator or HTTP tunneling.

9 IIOP over SSL

Using SSL to authenticate clients and to protect the communication between client and target requires no changes in your source code. The only notable effect is that SSL/TLS type sockets are used for transport connections instead of plain TCP sockets — and that connection setup takes a bit longer.

The only prerequisites are that you rebuild JacORB with cryptography support. You also need to set up a key store file that holds your cryptographic keys, and to configure SSL by setting a few properties. All of this is described in this chapter.

9.1 Re–Building JacORB's security libraries

In the standard distribution, the JacORB security libraries are not enabled. To do so, you simply need to recompile JacORB with the required SSL and JCE libraries in your CLASSPATH. If these libraries are not found, JacORB will be rebuilt without SSL support.

To successfully rebuild JacORB with SSL support, the following is required:

- JDK 1.2 or later,
- IAIK-JCE 2.591, 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.[1]

Make sure that the two libraries are in your CLASSPATH. Then build JacORB anew by typing `ant` in your JacORB installation directory.

9.2 Setting up a key store

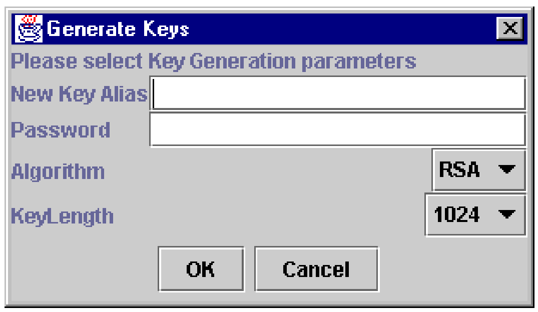
SSL relies on public key certificates in the standard X.509 format. These certificates are presented in the authentication phase of the SSL handshake and used to compute and exchange session keys. This section explains how to create and store these certificates.

[1] We will allow using SUN provider and SSL implementation in a future release

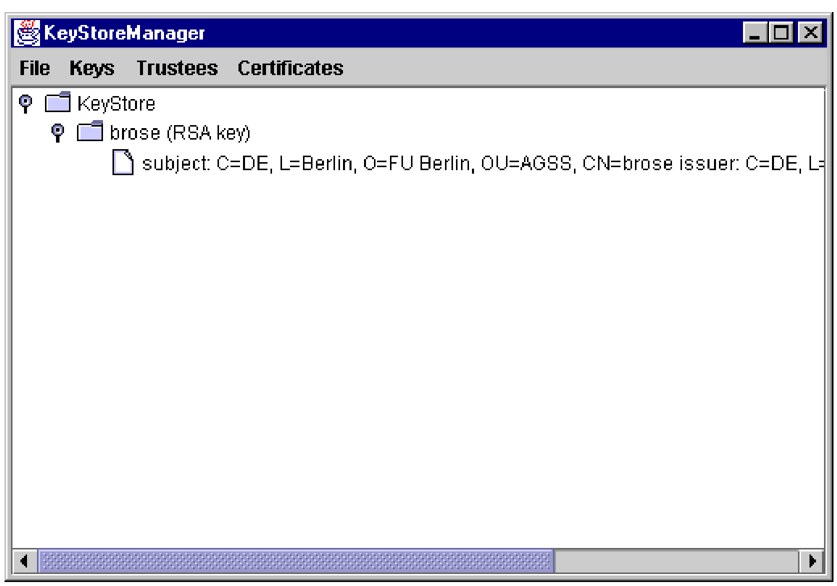
The Java 2 security API provides interfaces that access a persistent data structure called *KeyStore*. A key store is simply a file that contains public key certificates and the corresponding private keys. It also contains other certificates that can be used to verify public key certificates. All cryptographic data is protected using passwords and accessed using names called *aliases*.

JacORB provides a GUI tool to create and manipulate key store files, the KeyStoreManager. It can generate key pairs, sign public keys, import or export certificates, and define trusted certificate authorities. To start the KeyStoreManager, simply type `ks` on the command line. The GUI lets you select and open existing key store files, or create new ones.

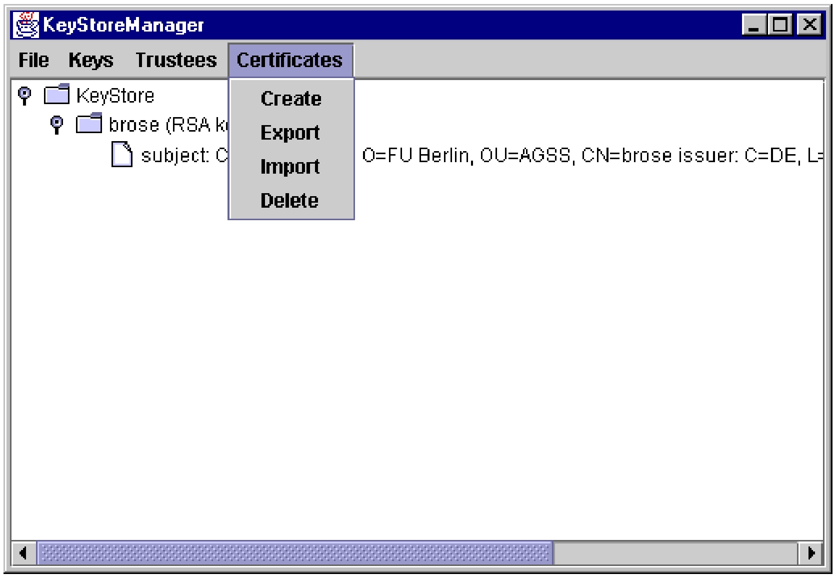
Starting with an empty key store, you first need to create a new key store and then a key pair and certificate. Select `New` from the `File` menu to create a key store, and then `New` from the `Keys` menu. You will then be asked to provide a new alias name for your new key entry. You also need to choose a password. You can leave the algorithm and key length fields in the combobox menu unchanged.



You now have a public key certificate that you can present for authentication, claiming identity with the alias name that has been embedded in the certificate. Since anybody could present such a certificate, receivers require that the certificate be digitally signed by someone they trust, a *Certificate Authority* (CA). By signing the certificate, a CA supports the identity claim of the certificate subject. Whose signature is accepted as trustworthy is just a matter of configuration, but normally proper CAs are expected to only sign certificates that they have carefully scrutinized — or even created themselves.



For convenience you can act as a CA yourself, using the KeyStoreManager GUI to import certificates and then sign and export them again. The originating key store can then re–import the certificate that now bears the digital signature of someone acting as a CA. The key store has a standard key chain format that must be used to store public key certificates. The first entry in the key chain is your own public key certificate as generated by the key store. It is automatically signed with its own private key. Second in the chain is the public key certificate that is signed by the CA. The last entry in a key chain must hold the CA's public key certificate, signed using its private key. Trust in the CA key is "axiomatic".



You can check the validity of a key chain by selecting an alias and then choosing `Verify Chain` from the `Keys` menu. Unless the key chain has the proper format *and* the CA's public key certificate is also declared as trusted using the `Trustees--add` menu, the verification will fail. Only of the verification succeeds will you be able to use a public key certificate in the SSL connection setup. More documentation on key stores can be found in the Java tool documentation for the `keytool` command. If you care for "real" security, be advised that setting up and managing (or finding) a properly administered CA is essential for the overall security of your system.

Finally, note that key stores are normally used only for client authentication in JacORB. Servers may, but need not, have their own keys and passwords because server authentication is optional and not mandatory like client authentication. Technically, this is achieved by exchanging the client and server roles at SSL setup. This is entirely transparent to applications, of course, but might prevent interoperation with other ORBs over SSL if their SSL setup is not prepared to handle this role change.

9.2.1 Step–By–Step certificate creation

In order to generate a simple public key infrastructure you can perform the following steps:

1. Create new keystores (File/new) and keypairs (Keys/new) for the CA and for the user.
2. Open the user keystore (File/open), select the key entry and export the self-signed certificate (Certificates/Export).

3. Open the CA keystore and add the user certificate as a Trustee (Trustees/add...).
4. Select the trusted user certificate and create a signed public key certificate (Certificates/Create). Leave the role name field empty, enter the CAs private key password and save the new certificate by clicking OK.
5. Export the CAs self-signed certificate to a file (as explained above). Delete the trusted certificate from the CA keystore (Trustees/Delete).
6. Open the user keystore again. Select the key entry, the import the CA-signed user cert (Certificates/Import), and the self-signed CA cert.
7. Add the self-signed CA cert as a trustee. This is only needed for verifying the chain, therefor the keystore can be deployed without it. Please note that a failed verification might result in a SignatureException.

9.3 Configuring SSL properties

When the ORB is initialized by the application, a couple of 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 `enforce_ssl` is on, your application requires that SSL be used for all its connections, both incoming and outgoing. `CORBA::NO_PERMISSION` is thrown if the policy is violated. If your application is a server, you can thus use SSL to authenticate your clients and, in more advanced scenarios using interceptors, perform operation–based access control. As a client, you would probably only require SSL if you are careful to protect your communication from eavesdropping or tampering.

The `support_ssl` option is used if you don't require protection yourself but are prepared to use SSL if the other side requires it. If `support_ssl=off` then `enforce_ssl=off` also. If `jacorb.security.support_ssl=on` the user will have to authenticate himself. To be able to provide the required certificates, the user will have to provide the key store alias and a password to the ORB.

These SSL settings can be further refined using security options as in the following property definitions:

```
jacorb.security.ssl.supported_options=40
jacorb.security.ssl.required_options=0
```

The value of these security options is a bit mask coded as a hexadecimal integer. The meanings of the individual bits is defined in the CORBA Security Service Specification and reproduced here from the `Security.idl` file:

```
typedef unsigned short   AssociationOptions;

const AssociationOptions NoProtection = 1;
const AssociationOptions Integrity = 2;
const AssociationOptions Confidentiality = 4;
const AssociationOptions DetectReplay = 8;
const AssociationOptions DetectMisordering = 16;
const AssociationOptions EstablishTrustInTarget = 32;
const AssociationOptions EstablishTrustInClient = 64;
const AssociationOptions NoDelegation = 128;
const AssociationOptions SimpleDelegation = 256;
const AssociationOptions CompositeDelegation = 512;
```

With the current SSL integration in JacORB, the only valid settings are EstablishTrustInTarget and/or EstablishTrustInClient, i.e. hex values 40 or 60. NoProtection is not possible when SSL is used. If you don't want protection, switch SSL support off.

As explained in the previous section, cryptographic data (key pairs and certificates) is stored in a keystore file. To configure the file name of the keystore file, you need to define the following property:

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

To avoid typing in lots of aliases and passwords (one for the key store, and one for each entry that is used), you can define default aliases and passwords like this:

```
# the name of the default key alias to look up in the keystore
jacorb.security.default_user=brose
jacorb.security.default_password=jacorb

# the name and location of the keystore relative to the home directory
jacorb.security.keystore=.keystore
```

10 Portable Interceptors

Since revision 1.1 JacORB provides support for Portable Interceptors These interceptors are compliant to the specification which can be found at http://cgi.omg.org/cgi-bin/doc?ptc/00-03-03. Therefore we don't provide any documentation on how to program interceptors but supply a few (hopefully helpful) hints and tips on JacORB specific solutions.

The first step to have an interceptor integrated into the ORB is to register an *ORBInitializer*. This is done by setting a property the following way:

```
org.omg.PortableInterceptor.ORBInitializerClass.<any_suffix>=
   <orb initializer classname>
```

The suffix is just to distinguish between different initializers and doesn't have to have any meaningful value. The value of the property however has to be the fully qualified classname of the initializer. If the verbosity is set to ≥ 2 JacORB will display a `ClassNotFoundException` in case the initializers class is not in the class path.

An example line might look like:

```
org.omg.PortableInterceptor.ORBInitializerClass.my_init=
   test.MyInterceptorInitializer
```

Unfortunately the interfaces of the specification don't provide any access to the ORB. If you need access to the ORB from out of the initializer you can cast the `ORBInitInfo` object to `jacorb.orb.portableInterceptor.ORBInitInfoImpl` and call `getORB()` to get a reference to the ORB that instantiated the initializer.

When working with service contexts please make sure that you don't use `java.lang.Integer.MAX_VALUE` as an id because a service context with that id is used internally. Otherwise you will end up with either your data not transfered or unexpected internal exceptions.

11 JacORB utilities

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

11.1 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>][-W debug_level] <filelist>
```

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

The `-noskel` option suppresses the generation of skeleton files, which may be unnecessary if you only want to *use* a reference and thus need client–side code.

The `-ir` option instructs the compiler to generate additional files that contain information needed by the Interface Repository. Basically, there is one (very small) extra file for each IDL module, and another additional file per IDL interface.

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. The `-U` option lets you undefine symbols again.

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`.)

11.2 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> [<timeout>]
```

or

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

Example

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

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 in the current directory unless the property `jacorb.naming.db_dir` is set to a different directory name. 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 overridden 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 specified in milliseconds.

11.3 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
```

11.4 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.

11.5 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
```

11.6 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>
```

11.7 ir

This command starts the JacORB Interface Repository, which is explained in chapter 7.

Usage

```
$ ir <reppository calss path> <IOR filename>
```

11.8 qir

This command queries the JacORB Interface Repository and prints out re–generated IDL for the repository item denoted by the argument repository ID.

Usage

```
$ qir <reppository Id>
```

11.9 ks

This command starts the JacORB KeyStoreManager, which is explained in chapter 9

Usage

```
$ ks
```

12 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

[BVD01] Gerald Brose, Andreas Vogel, and Keith Duddy. *Java Programming with CORBA*. John Wiley & Sons, 3rd edition, 2001.

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

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

[Sie00] Jon Siegel. *CORBA 3 Fundamentals and Programming*. Wiley, 2nd edition, 2000.

[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.