# JacORB 2.3.1 Programming Guide

The JacORB Team

December 20, 2010

**Contributors in alphabetical order:**

Alphonse Bendt
Gerald Brose
Nick Cross
Phil Mesnier
Nicolas Noffke
Steve Osselton
Simon McQueen
Francisco Reverbel
David Robison
André Spiegel

# 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. The JacORB version described in this document is JacORB 2.3.1.

## 1.1 A Brief CORBA introduction

CORBA models distributed resources as objects that provide a well-defined interface. CORBA lets you invoke services through remote invocations (RPCs). Since the transfer syntax for sending messages to objects is strictly defined, it is possible to exchange requests and replies between processes running program written in arbitrary programming languages and hosted on arbitrary hardware and operating systems. Target addresses are represented as *Interoperable Object References* (IORs), which contain transport addresses as well as identifiers needed to dispatch incoming messages to implementations.

Interfaces to remote objects are described declaratively in an programming language-independent *Interface Definition Language* (IDL), which can be used to automatically generate language-specific stub code.

It is important to stress that:

- CORBA objects as seen by clients are abstract entities. Their behavior is implemented by artifacts in potentially arbitrary, even non-OO languages. These artifacts are called *servants* in CORBA terminology. A servant is *not* the same as the object. Servants require an ORB implementation to maintain the relationship to objects and to mediate requests and responses.

- CORBA objects achieve location transparency, i.e., clients need not be (and generally are not) aware of the actual target hosts where servants reside. However, complete distribution transparency is not achieved in the sense that clients would not notice a difference between a local function call and a remote CORBA invocation. This is due to factors such as increased latency, network error conditions, and CORBA-specific initialization code in applications, and data type mappings.

Please see [**?**, **?**, **?**] for more information and additional details, and [**?**] for advanced issues.

## 1.2 Project History

JacORB originated in 1995 (was it 1996?) in the CS department at Freie Universität Berlin (FUB). It evolved from a small Java RPC library and a stub compiler that would process Java interfaces. This predecessor was written — most for fun and out of curiosity — by Boris Bokowski and Gerald Brose because at that time no Java RMI was available. The two of us then realized how close the Java interface syntax was to CORBA IDL, so we wrote an IDL grammar for our parser generator and moved to GIOP and IIOP as the transport protocol. It was shortly before Christmas 1996 when the first interoperable GIOP request was sent form a JacORB client to an IONA Orbix server. For a long time, JacORB was the only free (in the GNU sense) Java/CORBA implementation available, and it soon enjoyed widespread interest, at first mostly in academic projects, but commercial use followed soon after.

For a while, Gerald developed JacORB as a one-man-project until a few student projects and master theses started adding to it, most notably Reimo Tiedemann's POA implementation, and Nicolas Noffke's Implementation Repository and Portable Interceptor implementations. Other early contributors were Sebastian Müller, who wrote the Appligator, and Herbert Kiefer, who added a policy domain service. The Appligator and the policy domain service are no longer part of the JacORB distribution.

A more recent addition is Alphonse Bendt's implementation of the CORBA Notification Services as part of his master's theses. Substantial additions to the JacORB core were made by André Spiegel, who contributed OBV and AMI implementations. Other substantial contributions to JacORB have been added over time by the team at PrismTech UK (Steve Osselton, Nick Cross, Simon McQueen, Jason Courage). Still other active contributors are Francisco Reverbel of the JBoss team (RMI/IIOP), David Robison, who contributed CSIv2 and Phil Mesnier of OCI (http://www.ociweb.com).

JacORB continues to be used for research at FUB, especially in the field of distributed object security. Even though a number of people from the core team have left FUB; Gerald is with Projektron BCS (http://www.projektron.de), Reimo is with CoreMedia (http://www.coremedia.com), Nico and Alphonse are with Xtradyne (http://www.xtradyne.com) (now part of PrismTech (http://www.prismtech.com)) and André Spiegel is now a free-lance developer and consultant (http://www.free-software-consulting.com), the JacORB project is still rooted at Freie Universität Berlin, which hosts the JacORB web and CVS server.

Due to the limited number of developers, the philosophy around the development has never been to achieve feature-completeness beyond the core 90%, but standards compliance and quality. (e.g., JacORB 2.0 does not come with a PolicyManager). Brand-new and less widely-used features had to wait until the specification had reached a minimum maturity — or until someone offered project funding.

## 1.3 Support

The JacORB core team and the user community together provide best effort support over our mailing lists.

To enquire about commercial support, please send email to `info@jacorb.com` if you want members of the JacORB core team. Commercial support is also available from PrismTech and OCI.

## 1.4 Contributing — Donations

In essence, the early development years were entirely funded by public research. JacORB did receive some sponsoring over the years, but not as much as would have been desirable. A few development tasks that would otherwise not have been possible could be payed for, but more would have been possible — and still is.

If you feel that returning some of the value created by the use of Open Source software in your company is a wise investment in the future of that the software (maintenance, quality improvements, further development) in the future, then you should contact us about donations.

Buying hardware and sending it to us is one option. It is also possible to directly donate money to the JacORB project at Freie Universität Berlin. If approval for outright donations is difficult to obtain at your company, we can send you an invoice for, e.g.., CORBA consulting.

## 1.5 Contributing — Development

If you want to contribute to the development of the software directly, you should do the following:

- download JacORB and run the software to gain some first-hand expertise first
- read this document and other sources of CORBA documentation, such as [**?**], and the OMG's set of specifications (CORBA spec., IDL/Java language mapping)
- start reading the code
- subscribe to the `jacorb-developer` mailing list to share your expertise
- contact us to get subscribed to the core team's mailing list and gain CVS access
- read the coding guide line
- contribute code and test cases

## 1.6 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.

### 1.6.1 Feedback, Bug reports

For bug reporting, please use our Bugzilla bug tracking system available at http://www.jacorb.org/bugzilla. Please send problems as well as criticism and experience reports to our developer mailing list available from http://www.jacorb.org/contact.html.

# 2 Installing JacORB

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

## 2.1 Downloading JacORB

JacORB can be downloaded as a g-zipped tar–archive or as a zip–archive from the JacORB home page at http://www.jacorb.org.

To install JacORB, first unzip and untar (or simply unzip) the archive somewhere. This will result in a new directory `JacORB2_3_1`. After this follow the instructions in `JacORB2_3_1/doc/INSTALL`.

## 2.2 Installation

### 2.2.1 Requirements

JacORB requires JDK 1.5 or above properly installed on your machine. To build 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.

For SSL, you need an implementation of the SSL protocol. We currently support Oracle's JSSE Reference implementation included in the JDK.

### 2.2.2 Dependencies

JacORB depends upon the following third party software

1. Simple Logging Facade For Java (SLF4J Version 1.5.0)

# 3 Configuration

This chapter explains the general mechanism for configuring JacORB and lists all configuration properties. Note that as JacORB's configuration has been updated it is recommended to use the new jacorb.properties file supplied with this version.

## 3.1 Properties

JacORB has a number of configuration options which can be set as Java properties. There are three options for setting properties:

- in properties files
- as command line properties, and
- as properties passed as arguments to ORB.init() in the code of your applications.

In the case of a single JVM with multiple ORB instances, it may be required to either share configuration options between ORBs, or to separate the individual configurations from each other. We explain how properties can be set for sharing or for individual ORB instances.

### 3.1.1 Properties files

JacORB looks for a few standard properties files, a common file called `orb.properties`, and an ORB-specific file called `<orbid>.properties`, where `<orbid>` is the name of an ORB instance that was explicitly configured. Moreover, JacORB can load custom properties files from arbitrary locations. We explain each of these files in turn.

#### The common properties file

The reason for having a common properties file is that a single JacORB installation may be shared by a number of users with a set of common default properties. These may be refined by users in their own properties files but still provide reasonable defaults for the environment. Note

that it is not required to have a common properties file as all configuration options can also be set in other files, on the commandline or in the code.

JacORB looks for the common properties file `orb.properties` in the following places:

1. in the `lib` directory of the JDK installation. (The JDK's home directory denoted by the system property "java.home").
2. in the user home directory. (This is denoted by the system property "user.home". On Windows, this is `c:\documents\username`, on Unixes it's `~user`. If in doubt where your home directory is, write a small Java programm that prints out this property.
3. on the class path.

The common properties file is searched in the order presented above, so you may actually be loading multiple files of this name. If a properties file is found it is loaded, and any property values defined in this file will override values of the same property that were loaded earlier. Loading properties files from the classpath is useful when distributing applications packaged in JAR files.

### The ORB properties file

Having ORB-specific properties files is necessary when multiple ORB instances live in the same process, but need to have separate configurations, e.g., some ORBs use SSL and others don't, or some ORBs need to listen on separate but predefined ports. To let colocated ORBs use and retrieve separate configurations, JacORB provides a lookup mechanisms based on a specific property, the `ORBid` property. The default value for the ORBid is `jacorb`, ie. is the ORBid is not explicitly set anywhere, it defaults to `jacorb`. Note that this ORBid is reserved, ie., you cannot explicitly set your ORBid to this value. To use different configurations for different ORBs, you simply pass different ORBid values to your ORBs.

JacORB looks for ORB properties files in these places:

1. `jacorb.config.dir/etc/orbid.properties.`, if that exists, or
2. `jacorb.home/etc/orbid.properties.`, or
3. the current directory (`'./orbid.properties.'`)
4. on the class path.

The `jacorb.config.dir` and `jacorb.home` properties must be set for JacORB to be able to use a preconfigured configuration directory. The `jacorb.home` property defaults to

`''.''`, if unset. Setting these properties can be done in the `orb.properties` file, or by passing a property in on the commandline, like this:

```
$ jaco -Djacorb.config.dir=c:/ -DORBid=example test.Example
```

This commandline causes JacORB to look for a file called `example.properties` in `c:/etc`. If the `-DORBid=example` had been ommitted, the name of the ORB properties file that JacORB would try to load would have been `jacorb.properties`, because that is the default value for the ORBid. A good starting point is to have a common properties file that sets the `jacorb.config.dir` property, and then have put a `jacorb.properties` file in that directory.

Note, however, that the added flexibility of using multiple configuration files may lead to individual properties defined in multiple files. You must know the order in which your configuration files are loaded to avoid confusion over property settings not having the expected effect!

### Custom properties files

In addition to the standard JacORB properties files, a *custom properties file* can be loaded by passing the name of that properties files the `custom.props` property to JacORB. This can be handy for application-specific settings that you want to distribute with your code.

The value of this property is the path to a properties file, which contains the properties you want to load. As an example, imagine that you usually use plain TCP/IP connections, but in some cases want to use SSL (see section 11). The different ways of achieving this are

- Use just one properties file, but you will have to edit that file if you want to switch between SSL and plaintext connections.
- Use commandline properties exclusively (cf. below), which may lead to very long commands
- Use a command property file for all applications and different custom properties files for each application.

For example, you could start a JacORB program like this:

```
$ jaco -Dcustom.props=c:/tmp/ns.props org.jacorb.naming.NameServer
```

In addition to loading any standard properties files found in the places listed above, JacORB will now also load configuration properties from the file `c:/tmp/ns.props`, but this last file will be loaded after the default properties files and its values will thus take precedence over earlier settings.

### 3.1.2 Command-line properties

In the same way as the `custom.props` property in the example above, arbitrary other Java properties can be passed to JacORB programs using the `-D<prop name>=<prop value>` command line syntax for the `java` interpreter, but can be used in the same way with the `jaco` script. Note that the properties must precede the class name on the command line. For example to override the ORB initial references for NameService the following may be used:

```
jaco -DORBInitRef.NameService=file:///usr/users/...../NameService.ior
    Server
```

The ORB configuration mechanism will give configuration properties passed in this way precedence over property values found in configuration files.

Anything that follows after the class name is interpreted (by `java`) as a command line argument to the class and will be visible in the `args` parameter of the classes main method. For example

```
jaco Server
    -ORBInitRef.NameService=file:///usr/users/..../NameService.ior
```

### 3.1.3 Arguments to ORB.init()

For more application–specific properties, you can pass a `java.util.Properties` object to `ORB.init()` during application initialization. Properties set this way will override properties set by a properties file. The following code snippet demonstrates how to pass in a `Properties` object (`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);
```

## 3.2 Common Configuration Options

We are now ready to have a look at the most basic JacORB configuration properties. As a starting point, you should look at the file `/etc/jacorb_properties.template`, which you can adapt to your own needs (e.g. renaming to jacorb.properties or orb.properties as required).

### 3.2.1 Initial references

Initial references are object references that are available to CORBA application through the bootstrap `orb.resolve_initial_service()` API call. This call takes a string argument as the name of an initial reference and returns a CORBA object reference, e.g., to the initial name service.

```
########################################
#                                      #
#   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-POA/
#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
```

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 5).

### 3.2.2 Logging

JacORB writes logging information through SLF4J, which is a logging facade that can interface with arbitary backend logging systems such as Log4J, JCL, or JDK logging. To switch to a different logging system, all that needs to be done is to put a different library on the classpath.

JacORB does not usually attempt to configure the external logging system. That means it is left to you to provide a configuration for it, for example to set the log verbosity or to configure log file names. This is done in a way that is specific to the particular logging system used, e.g. via property files. As an added convenience, there is also a hook in JacORB that allows you to make some settings in the external logging system based on the configuration of JacORB itself, for example to choose log file names based on the implementation name of the server.

The default logging system selected in the JacORB distribution is JDK logging. This is because it is already present in the JDK, and JacORB therefore does not need to ship any other library. A sample configuration file for JDK logging is provided, which makes it easy to specify log file names, log rotation parameters, and the like.

### Logging Conventions

JacORB logs to a hierarchy of loggers with a root logger named `jacorb`. All sub-loggers have names that start with this prefix, e.g. `jacorb.orb`, `jacorb.config`, and so on. Settings that apply to the root logger usually also apply to all loggers below in the hierarchy (depending on the actual logging system used).

It is possible to split the logging hierarchy based on the implementation name of the ORB instance. Different ORBs will then log to different loggers, which can be configured independently. To do this, set the property `jacorb.log.split_on_implname` to `true`. Then, if the property `jacorb.implname` is set for an ORB instance, the loggers for that ORB all start with the prefix `jacorb.`$implname$`.`, rather than just `jacorb.`.

SLF4J defines five different logging levels: *error*, *warn*, *info*, *debug*, and *trace*. These levels are mapped to the log levels of the underlying logging system. For JDK logging, the mapping is *error* → `SEVERE`, *warn* → `WARNING`, *info* → `INFO`, *debug* → `FINE`, and *trace* → `FINEST`. In JacORB's code, the SLF4J log levels are used according to the following conventions:

**error** Events which suggest that there is a bug in JacORB or in user code. This includes, but is not limited to, "fatal errors" which will lead to termination of the program.

**warn** Events that demand attention, but which are handled properly according to the CORBA spec. For example, abnormal termination of a connection, reaching of a resource limit (queue full), and the like.

**info** Starting and stopping of subsystems, establishing and closing of connections, registering objects with a POA.

**debug** Information that might be needed for finding bugs in JacORB or user code. Anything that relates to the normal processing path of individual messages.

**trace** Not used in JacORB (and discouraged by the SLF4J team).

### Configuration of JDK Logging

Since JDK logging is the default in the JacORB distribution, we provide some shortcuts to achieve a reasonable logging configuration with it easily. If any of the two properties `jacorb.log.default.verbosity` or `jacorb.logfile` is set, then JacORB configures JDK logging at startup to match these values. (These properties were retained from previous JacORB versions where JacORB configured all logging itself.)

The property `jacorb.log.default.verbosity` specifies the level at which messages are logged. The value is a number from 0 to 4, where 0 means no logging (off), 1 means only $error$ messages, 2 means $warn$ messages, 3 means $info$ messages, and 4 means $debug$ messages (higher levels also include lower levels).

The property `jacorb.logfile` specifies the name of a file to write the log to. If this property is not set, then logging goes to the console. You may include the string "$implname" in the value of this property; this is replaced with the value of the property `jacorb.implname` if that is set, or "jacorb" otherwise.

The default formatting of JDK logs is quite verbose, using two lines of output for every log entry. We have provided a `LogFormatter` class that gives a more succinct output. This class is named `org.jacorb.config.JacORBLogFormatter` and is used whenever JacORB configures the logging system itself.

For any more sophisticated configuration, such as using log file rotation or specifying different log levels for some of the loggers, you need to configure JDK logging directly. To do this, make sure that the properties `jacorb.log.default.verbosity` and `jacorb.logfile` are *not* set to any value (because otherwise JacORB will interfere with your configuration). Then, prepare a configuration file such as the one found in `JACORB_HOME/etc/logging_properties.template` with your settings. The name of this file needs to be passed to the JVM at startup, for example by providing the following command line option:

```
-Djava.util.logging.config.file=/path/to/config-file
```

### Using another Logging System

The SLF4J facade is implemented in two Java libraries, a generic one and a backend-specific one. The generic library is named `slf4j-api-1.5.10.jar` (or any other version), and the backend-specific library is named `slf4j-jdk14-1.5.10.jar` (for JDK logging), or `slf4j-log4j-1.5.6.jar` for Log4J, and the like. To switch to a different logging system, the backend-specific library of SLF4J needs to be replaced, and the implementation library for that backend needs to be added as well. So for example, to use Log4J, the following libraries need to be on the classpath: `slf4j-api-1.5.10.jar`, `slf4j-log4j-1.5.10.jar`, and `log4j-1.2.15.jar`. The JacORB distribution only ships with the generic SLF4J library and the JDK adapter library. Other libraries need to be downloaded from the SLF4J site.

When using another logging system besides JDK logging, JacORB does not attempt to configure log verbosity or log file names by itself, as described in the previous section. This means that features such as choosing log file names based on implementation names are not available for other logging systems. You can however configure this explicitly, for example by using the split-on-implname feature described above. For more sophisticated needs, it is also possible to provide a `LoggingInitializer` class, which is a hook provided by JacORB to allow configuration of a logging system based on the JacORB configuration. The class needs to extend the class `org.jacorb.config.LoggingInitializer` and the name of the class needs to be specified in the property `jacorb.log.initializer`.

### POA Monitor

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 6 on the POA for more details.

## 3.3 Configuration Properties

A comprehensive listing and description of the properties which are used to configure JacORB are given in the following tables.

Table 3.1: ORB Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| `ORBInitRef.<service>` | Properties of this form configure initial service objects which can be resolved via the ORB resolve_initial_references. A variety of URL formats are supported. | URL | unset |
| `org.omg.PortableInterceptor.ORBInitializerClass.<name>` | A portable interceptor initializer class instantiated at ORB creation. | class | unset |
| `jacorb.orb.objectKeyMap.<name>` | Maps an object key to an arbitrary string thereby enabling better readability for corbaloc URLs. | string | |
| `jacorb.giop_minor_version` | The GIOP minor version number to use for newly created IORs | integer | 2 |
| `jacorb.retries` | Number of retries if connection cannot directly be established | integer | 5 |
| `jacorb.retry_interval` | Time in milliseconds to wait between retries | millisec. | 500 |

Table 3.1: ORB Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| `jacorb.buffermanager.factory` | This parameter allow to define buffer manager factory. Here are 3 options already implemented:<br>1. org.jacorb.orb.DefaultBufferManagerFactory that will create default buffer manager implementation<br>2. org.jacorb.orb.JDK15BufferManagerFactory that uses JDK 1.5 (or above) buffer manager implementation based on the soft references (java.lang.ref.SoftReference).<br>3. org.jacorb.orb.NonCachingBufferManagerFactory that uses simple buffer manager implementation without any caching.<br>Also, custom-made buffer manager factories allowed. They must implement the org.jacorb.orb.BufferManagerFactory interface. | class | org.jacorb.orb.Defa |
| `jacorb.maxManagedBufSize` | This is NOT the maximum buffer size that can be used, but just the largest size of buffers that will be kept and managed. The real value of the maximal managed buffer size in bytes is (2**maxManagedBufSize). You only need to increase this value if you are dealing with LOTS of LARGE data structures. You may decrease it to make the buffer manager release large buffers immediately rather than keeping them for later reuse | integer | 22 |
| `jacorb.bufferManagerFlushMax` | Whether to use an additional unlimited size buffer cache for CDROutputStreams. If -1 then off, if zero then this is feature is enabled, if greater than zero then it is enabled and flushed every x seconds | integer | -1 |
| `jacorb.bufferManagerThreshold` | Maximum number of buffers of the same size held in pool. | integer | 20. |

Table 3.1: ORB Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| `jacorb.buffermanager.expansionpolicy` | This parameter allow to define buffer manager expansion policy. Here are 3 options already implemented:<br>1. org.jacorb.orb.buffermanager.DefaultExpansionPolicy that will return new buffer's size that bigger or equal to the requested. Sizes calculation are performed by code:<br>`double multiplier = scale - Math.log (reque`<br>`multiplier = (multiplier < 1.0) ? 1.0 : mul`<br>`newSize = (int) Math.floor ( multiplier * r`<br>where scale and divider parameters are configurable (see description below).<br>2. org.jacorb.orb.buffermanager.LinearExpansionPolicy that returns exactly requested size.<br>3. org.jacorb.orb.buffermanager.DoubleExpansionPolicy that returns new size wich equals requested size * 2.<br>Also, custom-made buffer manager expansion policies are allowed. They must implement the org.jacorb.orb.buffermanager.BufferManagerExpansionPolicy interface. Please note that expansion policy support is implemented in the default buffer manager implementation (org.jacorb.orb.BufferManager). Custom-made buffer manager implementation need to have their own expansion policy support implementation. | class | org.jacorb.orb.buffe |
| `jacorb.buffermanager.defaultexpansionpolicy.scale` | Scale parameter for the org.jacorb.orb.buffermanager.DefaultExpansionPolicy buffer sizes calculation (see the formula above). | float | 4 |
| `jacorb.buffermanager.defaultexpansionpolicy.divider` | Divider parameter for the org.jacorb.orb.buffermanager.DefaultExpansionPolicy buffer sizes calculation (see the formula above). | float | 6 |
| `jacorb.deferredArrayQueue` | JacORB will delay internally transferring bytes to the stream; this is the size of this internal queue. Size in k. | integer | 8. |

Table 3.1: ORB Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| jacorb.connection.delay_close | Normally, a jacorb server will close the TCP/IP connection right after sending a CloseConnection message. However, it may occasionally happen that the client sends a message into the closed connection because it hasn't handled the CloseConnection yet. To avoid this situation, closing of the TCP/IP connection can be delayed (Delay time is controlled by jacorb.connection.timeout_after_closeconnection specified in msecs) | boolean | off |
| jacorb.connection.client.connect_timeout | Initial timeout for establishing a connection. | millisec | 90000 |
| jacorb.connection.client.pending_reply_timeout | Wait the specified number of msecs for a reply to a request. If exceeded, a org.omg.CORBA.TIMEOUT exception will be thrown. Not set by default | millisec. | 0 |
| jacorb.connection.client.idle_timeout | Client-side timeout. This is set to non-zero in order to close the connection after specified number of milliseconds idle time. Only connections that don't have pending messages are closed, unless jacorb.connection.client.timeout_ignores_pending_messages is turned on. | millisec. | unset |
| jacorb.connection.client.timeout_ignores_pending_messages | Controls if client-side idle timeouts take care of pending messages or not. If "on", the connection is closed regardless of any pending messages, and all pending messages are cancelled (resulting in a COMM_FAILURE, unless jacorb.connection.client.retry_on_failure is turned on). | boolean | off |
| jacorb.connection.client.retry_on_failure | Controls if network failures on existing connections should yield a COMM_FAILURE or should trigger a remarshaling of all pending messages. Note that this should only be used with idempotent operations because the client side ORB has no way of knowing the processing state of the lost request on the server. | boolean | |
| jacorb.connection.server.timeout | Maximum time in milliseconds that a server keeps a connection open if nothing happens | millisec. | unset |

Table 3.1: ORB Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| `jacorb.connection.server.keepalive` | Enable SO_KEEPALIVE on server sockets. If the OS keepalive detects a TCP/IP connection to be broken, the effect is the same as if the TCP/IP connection has been closed gracefully. | boolean | false |
| `jacorb.connection.client.keepalive` | Enable SO_KEEPALIVE on client sockets. If the OS keepalive detects a TCP/IP connection to be broken, the effect is the same as if the TCP/IP connection has been closed gracefully.All pending replies will receive a `COMM_FAILURE`. | boolean | false |
| `jacorb.connection.max_server_connections` | This property sets the maximum number of TCP/IP connections that will be listened on by the server–side ORB. Only effective in conjunction with the other connection management properties. Please see 17.2. | integer | unlimited |
| `jacorb.connection.wait_for_idle_interval` | This property sets the interval to wait until the next try is made to find an idle connection to close. Only effective in conjunction with the other connection management properties. Please see 17.2. | millisec | 500 |
| `jacorb.listener.server_socket_timeout` | Sets a timeout on the (SSL) server socket. This is a workaround for JDK 1.3 on linux where a thread blocked on `accept()` isn't notified when closing that socket. Default is 0, i.e. off. See Java bug #4344135. NOTE: This is only useful in conjunction with the SI&C SSL socket factories. | millisec | 0 |
| `jacorb.connection.selection_strategy_class` | This property sets the `SelectionStrategy`. Only effective in conjunction with the other connection management properties. Please see 17.2. | class | |
| `jacorb.connection.statistics_provider_class` | This property sets the `StatisticsProvider`. Only effective in conjunction with the other connection management properties. Please see 17.2. | class | |
| `jacorb.connection.delay_close` | This property controls the behaviour after sending a GIOP CloseConnection messsage. If set to "on", the TCP/IP connection won't be closed directly. Instead, it is waited for the client to do so first. Please see 17.2. | boolean | off |

Table 3.1: ORB Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| `jacorb.listener.server_socket_timeout` | Sets a timeout on the (SSL) server socket. This is a workaround for JDK 1.3 on linux where a thread blocked on `accept()` isn't notified when closing that socket. Default is 0, i.e. off. See Java bug #4344135. NOTE: This is only useful in conjunction with the SI&C SSL socket factories. | millisec | 0 |
| `jacorb.transport.factories` | This property controls which transport plug-ins are available to the ORB. The value is a list of classes that implement the ETF `Factories` interface. | comma-separated list of classes | |
| `jacorb.transport.server.listeners` | Controls which transports should be offered by JacORB on the server side. The value is a list of numeric profile tags for each transport that should be available on the server side. | comma-separated list of integers | |
| `jacorb.transport.client.selector` | Name of a class that selects the transport profile to use for communication on the client side. The value is the fully qualified name of a class that implements `org.jacorb.orb.ProfileSelector`. | class | |
| `jacorb.reference_caching` | Whether or not JacORB caches objects references | boolean | unset |
| `jacorb.hashtable_class` | The following property specifies the class which is used for reference caching. WeakHashtable uses WeakReferences, so entries get garbage collected if only the Hashtable has a reference to them. This is useful if you have many references to short-living non-persistent CORBA objects. It is only available for java 1.2 and above. On the other hand the standard Hashtable keeps the references until they are explicitly deleted by calling _release(). This is useful for persistent and long-living CORBA objects | class | Hashtable |
| `jacorb.use_bom` | Use GIOP 1.2 byte order markers, since CORBA 2.4-5 | boolean | off |
| `jacorb.giop.add_1_0_profiles` | Add additional IIOP 1.0 profiles even if using IIOP 1.2 | boolean | off |
| `jacorb.dns.enable` | Use DNS names in IORs, rather than numeric IP addresses | boolean | off |
| `jacorb.dns.eager\_resolve` | resolve DNS names in IORs eagerly | boolean | on |

Table 3.1: ORB Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| jacorb.compactTypecodes | Whether to send compact typecodes. Options are 0 (off), on (full compaction of all optional parameters) | boolean | on |
| jacorb.cacheTypecodes | Whether to cache read typecodes | boolean | off |
| jacorb.cachePoaNames | Whether to cache poa names as an optimisation to save reparsing portions of the object key | boolean | off |
| jacorb.orb_initializer.fail_on_error | Control, if failing ORBInitializers should make the complete ORB.init() fail. | boolean | off |
| jacorb.acceptor_exception_listener_class | A class implementing interface org.jacorb.orb.listener.AcceptorExceptionListener. The implementation will be notified of any exception caught by the thread doing the ServerSocket.accept() and has the chance of taking appropriate action, e.g. shutting down the ORB. The default implementation will shutdown the ORB on all Errors and SSLExceptions. | String (class-name) | org.jacorb.orb.listener.Default |
| jacorb.interop.indirection_encoding_disable | Turn off indirection encoding for repeated typecodes. This fixes interoperability with certain broken ORB's eg. Orbix 2000 | boolean | off |
| jacorb.interop.comet | Enable additional buffer length checking and adjustment for interoperability with Comet CORBA/COM bridge which can incorrectly encode buffer lengths | boolean | off |
| jacorb.interop.lax_boolean_encoding | Treat any non zero CDR encoded boolean value as true (strictly should be 1 not non zero). This is useful for ORBs such as VisiBroker and ORBacus | boolean | off |
| org.omg.PortableInterceptor.ORBInitializerClass.bidir_init | This portable interceptor must be configured to support bi-directional GIOP | class | unset |

Table 3.1: ORB Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| `jacorb.ior_proxy_address` | Used to supply an alternative endpoint in locally created object references. This is intended for servers that export IORs for access from outside a firewall. The general form of the value is `<protocol>://<address>`. The protocol name in the value must match the protocol(s) used by the server. For example: `iiop://myhost:1234`. The given address is inserted into every IOR that the local ORB produces, without any check whether the address is valid, except that the protocol must be supported by the ORB, and the address must be parsable for that protocol. This property supercedes jacorb.ior_proxy_host and jacorb.ior_proxy_port. | string | unset |
| `jacorb.ior_proxy_host` | The properties jacorb.ior_proxy_host and jacorb.ior_proxy_port have been superceded by jacorb.ior_proxy_address (see above), which is a protocol-independent way of specifying endpoint addresses. The host/port properties are still recognized, but if jacorb.ior_proxy_address is specified, it overrides these properties. Note that the value that ends up in the IOR also is affected by the setting of the property jacorb.dns.enable. | node | unset |
| `jacorb.ior_proxy_port` | See jacorb.ior_proxy_host and jacorb.ior_proxy_address above | port | unset |
| `OAAddress` | Used to supply an explicit listener protocol and address for servers. The general form of the value is `<protocol>://<address>`. The protocol name must match the protocol(s) used by the server. For example: `iiop://myhost:1234`. This property supercedes OAIAddr and OAPort. | string | unset |

Table 3.1: ORB Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| `OAIAddr` | The Object Adapter Internet Address: IP address on multi-homed host (this gets encoded in object references).<br>• Addresses like 127.0.0.X will only be accessible from the same machine!<br>• If OAIAddr is not set on a multi-homed host it is operating system/JVM dependant which IP address is selected.<br>• If the developer is trying to use callbacks (*not bidirectional GIOP)* on a multihomed host the client will also require OAIAddr set as it is acting as a server. | node | unset |
| `OAPort` | See OAIAddr above (ignored if OAAddress is set) | port | unset |
| `org.omg.PortableInterceptor.ORBInitializerClass.standard_init` | Standard portable interceptor. DO NOT REMOVE. | class | |
| `jacorb.net.socket_factory` | Sets or defines the socket factory. See section 3.3.5 for details. | class | |
| `jacorb.net.server_socket_factory` | Sets or defines the server socket factory. See section 3.3.5 for details. | class | |
| `jacorb.net.socket_factory.port.min` | Sets the minimum port number that can be used for an additional supported socket factory. This property is used in conjunction with the jacorb.net.socket_factory.port.max property. These properties enable the factory to traverse firewalls through a fixed port range | integer | unset (disabled) |
| `jacorb.net.socket_factory.port.max` | Sets the maximum port number that can be used for the additional supported socket factory. Refer to jacorb.net.socket_factory.port.min above | integer | disabled |
| `jacorb.net.tcp_listener` | Defines a listener for TCP connection events. See 3.3.4. | string | disabled |
| `jacorb.enhanced_thread_name` | Temporarily adds connection endpoints and time (in milliseconds) that the thread started to the Thread name. To be used to correlate running threads with entries in debug logs. | string | off |

Table 3.1: ORB Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| `jacorb.avoidIsARemoteCall` | Always attempt to search for local repository ID information to avoid the cost of a remote call. In most scenarios this is quicker than the remote call. With some complicated hierarchies it may be quicker to turn this off. | boolean | on |
| `jacorb.native_char_codeset` | Overrides the detection from the local environment for the codeset used to transmit characters. Note that this property is only effective once per JVM. | string | off |
| `jacorb.native_wchar_codeset` | Overrides the detection from the local environment for the codeset used to transmit wide characters. ote that this property is only effective once per JVM. | string | off |
| `jacorb.codeset` | Enabling this will do codeset translation on marshalling. Disabling it will force JacORB to ignore all codeset component info profiles and to disable translation on marshalling. | boolean | on |

**Note:** The class `org.jacorb.orb.giop.CodeSet` provides a main method to aid debugging of codeset issues. It will print out the current system encoding values. If the developer is running under a Unix based system and passes the argument -a it will also print out the current locale and all known locales.

Table 3.2: Logging Configuration

| Property | Description | Type | Default |
|---|---|---|---|
| `jacorb.orb.print_version` | If enabled, the ORB's version number is printed whenever the ORB is initialized. | boolean | on |
| `jacorb.log.default.verbosity` | Log levels: 0 = off, 1 = error, 2 = warning, 3 = info, 4 = debug | integer | unset |
| `jacorb.logfile` | Output destination for diagnostic log file. If not set, diagnostics are sent to standard error. | filename | unset |
| `jacorb.logfile.append` | Whether to append to existing log file or overwrite (if file logging) | boolean | off |
| `jacorb.log.initializer` | Name of a class to initialize logging after the configuration has been read | class | `JdkLoggingInitializer` |
| `jacorb.debug.dump_outgoing_messages` | Hex dump outgoing messages | boolean | off |
| `jacorb.debug.dump_incoming_messages` | Hex dump incoming messages | boolean | off |

Table 3.3: Name service Configuration

| Property | Description | Type |
|---|---|---|
| jacorb.naming.purge | Whether non-active references are purged from name service when list operation is invoked. Default is off | on or off |
| jacorb.naming.noping | Whether resolve should return references without trying to ping them to see if they're still alive first. Default is ping (off) | on or off |
| jacorb.naming. ior_filename | The file where the name server drops its IOR (default unset) | string |

Table 3.4: POA Configuration

| Property | Description | Type |
|---|---|---|
| jacorb.poa.monitoring | Displays a GUI monitoring tool for servers. Default is off. | boolean |
| jacorb.poa.thread_pool _max | Maximum thread pool configuration for request processing | integer |
| jacorb.poa.thread_pool _min | Minimum thread pool configuration for request processing | integer |
| jacorb.poa.thread_pool _shared | If set use shared thread pool between all POAs. Only with ORB_CTRL_MODEL. Default is off. | boolean |
| jacorb.poa.thread_prio rity | If set, request processing threads in the POA will run at this priority. If not set or invalid, MAX_PRIORITY will be used. Not set by default. | integer |
| jacorb.poa.queue_wait | Specifies whether the POA should block when the request queue is full (On), or throw TRANSIENT exceptions (Off). Default is Off. | boolean |
| jacorb.poa.queue_max | The maximum length of the request queue. If this length has been reached, and further requests arrive, jacorb.poa.queue_wait specifies what to do. Default is 100. | integer |
| jacorb.poa.queue_min | If jacorb.poa.queue_wait is On, and the request queue gets full, then the POA blocks until the queue contains no more than queue_min requests. Default is 10. | integer |

Table 3.5: Implementation Repository Configuration

| Property | Description | Type |
|---|---|---|
| jacorb.use_imr | Switch on to contact the Implementation Repository (IR) on every server start-up. Default is off. | boolean |
| jacorb.use_imr_endpoin t | Switch off to prevent writing the IMR address into server IORs. This property is ignored if jacorb.use_imr = off. Default is off. | boolean |

Table 3.5: Implementation Repository Configuration

| Property | Description | Type |
|---|---|---|
| jacorb.imr.allow_auto_register | If set to on servers that don't already have an entry on their first call to the IR, will get automatically registered. Otherwise, an UnknownServer exception is thrown. Default is off. | boolean |
| jacorb.imr.check_object_liveness | If set on the IR will try to ping every object reference that it is going to return. If the reference is not alive, then TRANSIENT is thrown. Default is off. | boolean |
| ORBInitRef.ImplementationRepository | The initial reference for the IR. | URL |
| jacorb.imr.table_file | File in which the IR stores data. | file |
| jacorb.imr.backup_file | Backup data file for the IR. | file |
| jacorb.imr.ior_file | File to which the IR writes its IOR. This is usually referred to by the initial reference for the IR (configured above). | file |
| jacorb.imr.timeout | Time in milliseconds that the implementation will wait for a started server to register. After this timeout is exceeded the IR assumes the server has failed to start. Default is 12000 (2 minutes). | millisec. |
| jacorb.imr.no_of_poas | Initial number of POAs that can be registered with the IR. This is an optimization used to size internal data structures. This value can be exceeded. Default is 100. | integer |
| jacorb.imr.no_of_servers | Initial number of servers that can be registered with the IR. This is an optimization used to size internal data structures. This value can be exceeded. Default is 5. | integer |
| jacorb.imr.port_number | Starts the IMR on a fixed port (equivalent to the -p option). | integer |
| jacorb.imr.connection_timeout | Time in milliseconds that the IR waits until a connection from an application client is terminated. Default is 2000. | millisec. |
| jacorb.implname | The implementation name for persistent servers. See 3.3.1. | name |
| jacorb.java_exec | Command used by the IR to start servers. | command |

Table 3.6: Security Configuration

| Property | Description | Type |
|---|---|---|
| OASSLPort | The port number used by SSL, will be dynamically assigned by default. | port |
| org.omg.PortableInterceptor.ORBInitializerClass.ForwardInit | Portable interceptor required to support SSL. Not set by default. | class |
| jacorb.security.access_decision | The qualified classname of access decision object. | class |

Table 3.6: Security Configuration

| Property | Description | Type |
|---|---|---|
| jacorb.security.principal_authenticator | A list of qualified classnames of principle authenticator objects, separated by commas (no whitespaces.). The first entry (that can be successfully created) will be available through the principal_authenticator property. | class |
| jacorb.ssl.socket_factory | The qualified classname of the SSL socket factory class. See section 3.3.5 for details. | class |
| jacorb.ssl.server_socket_factory | The qualified classname of the SSL server socket factory class. See section 3.3.5 for details. | class |
| jacorb.security.support_ssl | Whether SSL security is supported. Default is off. | boolean |
| jacorb.security.ssl.client.supported_options | SSL client supported options - IIOP/SSL parameters (numbers are hex values, without the leading 0x): NoProtection = 1, EstablishTrustInClient = 40, EstablishTrustInTarget = 20, mutual authentication = 60. Default is 0. Please see the programming guide for more explanation. | integer |
| jacorb.security.ssl.client.required_options | SSL client required options (See IIOP/SSL parameters above). Default is 0. | integer |
| jacorb.security.ssl.server.supported_options | SSL server supported options (See IIOP/SSL parameters above). Default is 0. | integer |
| jacorb.security.ssl.server.required_options | SSL server required options (See IIOP/SSL parameters above). Default is 0. | integer |
| jacorb.security.ssl.corbaloc_ssliop.supported_options | Used in conjunction with jacorb.security.ssl.corbaloc_ssliop.required_options. If these properties are set, then two values will be placed in the IOR, ”corbaloc:ssliop” and ”ssliop”. If not set, only EstablishTrustInTarget is used for both supported and required options. | integer |
| jacorb.security.ssl.corbaloc_ssliop.required_options | Default is 0. | integer |
| jacorb.security.ssl.always_open_unsecured_endpoint | Default is FALSE. The secure interoperabilty spec states that targets that require SSL shall not open (or publicise in their IORs) an unsecured listen port. Some ORBs (we’re looking at you, MICO) apparently don’t like this. Setting this switch to TRUE will override the correct behaviour for interoperability. Attempts to access the unsecured port should be met with a NO_PERMISSION exception. | boolean |

Table 3.6: Security Configuration

| Property | Description | Type |
|---|---|---|
| jacorb.security.keystore | The name and location of the keystore. This may be absolute or relative to the home directory. NOTE (for Sun JSSE users): The javax.net.ssl.trustStore [Password] properties doesn't seem to take effect, so you may want to add trusted certificates to normal keystores. In this case, please set the property jacorb.security.jsse.trustees_from_ks to on, so trusted certificates are taken from the keystore instead of a dedicated truststore. | file |
| jacorb.security.keystore_password | The keystore password. | string |
| jacorb.security.keystore_type | The SSL keystore type. Defaults to JKS. | string |
| jacorb.security.jsse.server.key_manager_algorithm | The algorithm used to initialise the SSL socket factories. Defaults to SunX509. Change to IbmX509 for IBM JDKs. | string |
| jacorb.security.jsse.server.trust_manager_algorithm | The algorithm used to initialise the SSL socket factories. Defaults to SunX509. Change to IbmX509 for IBM JDKs. | string |
| jacorb.security.jsse.client.key_manager_algorithm | The algorithm used to initialise the SSL socket factories. Defaults to SunX509. Change to IbmX509 for IBM JDKs. | string |
| jacorb.security.jsse.client.trust_manager_algorithm | The algorithm used to initialise the SSL socket factories. Defaults to SunX509. Change to IbmX509 for IBM JDKs. | string |
| jacorb.security.jsse.trustees_from_ks | Sun JSSE specific settings: Use the keystore to take trusted certificates from. Default is off. | boolean |
| jacorb.security.ssl.server.cipher_suites | A comma-separated list of cipher suite names which must NOT contain whitespaces. See the JSSE documents on how to obtain the correct cipher suite strings. | string |
| jacorb.security.ssl.client.cipher_suites | See jacorb.security.ssl.server.cipher_suites above. | string |
| jacorb.security.randomclassPlugin | Classname for secure random plugin. See 11.3 | string |
| jacorb.security.ssl.ssl_listener | Defines a listener for SSL connection events. See 3.3.4. | string |

### Acceptor Exception Event Plugin

This plugin is implemented by `org.jacorb.orb.listener.AcceptorExceptionListener`.

```
package org.jacorb.orb.listener;
```

```
public interface AcceptorExceptionListener extends EventListener
    void exceptionCaught(AcceptorExceptionEvent ae);
```

The configuration property is

`jacorb.acceptor_exception_listener`

If the server listener thread receives an exception while doing the `ServerSocket.accept()` it will construct a `org.jacorb.orb.listener.AcceptorExceptionEvent` and notify the configured implementation. The Event allows the following to be retrieved:

```
public ORB getORB()
public Throwable getException()
```

The default implementation, `org.jacorb.orb.listener.DefaultAcceptorExceptionListener`, will simply shutdown the ORB on all Errors and for SSLExceptions that are thrown before any socket connections have been made. If the developer wishes they may plugin their own for more fine grained control.

In order to detect whether the exception has been thrown on the first attempt or any attempt after that the developer may use the following function within their listener implementation.

```
public void exceptionCaught(AcceptorExceptionEvent ae) {
...
   if (((org.jacorb.orb.iiop.IIOPListener.Acceptor)
               ae.getSource()).getAcceptorSocketLoop()) {
  ...
```

`getAcceptorSocketLoop` returns false if the event has been thrown on the initial loop, or true on any loop after that.

Note that if the default implementation is used it is possible that due to e.g. an SSLException the listener will fail to accept on the server socket after the root POA is resolved which means that the ORB will be shutdown. Therefore future calls on that POA will fail with a 'POA destroyed' message.

### 3.3.1 JacORB Implname and CORBA Objects

A JacORB object key consists of `<impl name>/<poa name>/<object oid>`. The lifespan of CORBA objects are defined by the POA policy LifespanPolicyValue.

Transient objects are those whose lifespans are bounded by the process in which they were created. Once a transient object has been destroyed any clients still holding references to those

objects should receive a OBJECT_NOT_EXIST. This applies even if the transient object is recreated as it is a new object reference. To achieve this JacORB replaces the implname portion of the key with transient data.

Persistent objects are those that may live beyond the lifetime of the process that created them. The implname property should be configured in this case. It should be set to a unique name to to form part of the object identity. If it is not set, an exception will be thrown. This property may be configured in the jacorb.properties (where an example shows it set to StandardImplName) or in the code of the server e.g.

```
/* 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);
```

The implname property allows a program to run with a different implementation name so that it will not accept references created by another persistent POA with the same POA name. A common problem is where the developer has two persistent servers running with the same implname and POA names when one tries to contact the other. Rather than calling server x, server y performs local call. This is because there is no way of distinguishing the two servers; the developer should have used different implnames (e.g. UUIDs).

### 3.3.2 Corbaloc Strings

Normally corbaloc is used to provide a shortcut to refer to CORBA objects. However the stringified key portion corresponds to the octet sequence in the object_key member of a GIOP Request or LocateRequest header as defined in section 15.4 of CORBA 2.3. Further the key_string uses the escape conventions described in RFC 2396 to map away from octet values that cannot directly be part of a URL. This means the key string might look like:

```
corbaloc:iiop:10.1.0.4:18000/FooBar/ServiceName/V_3%f1%1c%9b%11%db%b7%e9
         %bdsnQ%ea%85qV_3%f0%1c%9b%11%db%b7%e9%bdsnQ%ea%85TA5%f0%1c%9b%11
         %db%b7%e9%bdsnQ%ea%85
```

With JacORB, for persistent objects, the developer may configure the implname, poa name and object key. This should mean that the corbaloc sequence should be more readable:

```
corbaloc:iiop:10.1.0.4:42811/imr_demo/ImRDemoServerPOA/imr_demo
```

With a transient object the key may look like:

```
corbaloc:iiop:10.1.0.4:42818/2649480905/%00%14%3e45%0d%0b!%10%3e
```

As it is not possible to construct a transient object with a readable key some developers may find it useful to use the objectKeyMap facility within JacORB to refer to their transient objects. Note the objectKey functionality may also be used with persistent objects.

This property provides more readable corbaloc URLs by mapping the actual object key to an arbitrary string. The mapping below would permit clients of a name service to access it using corbaloc::ipaddress:portnum/NameService. The property also accepts the following mappings:

- IOR, resource, jndi, URL (e.g. file, http)

Note that `jacorb.orb.objectKeyMap.name` is configurable both through the jacorb.properties file and through the proprietary function

```
ORB::addObjectKey(String name, String)
```

Example usage

```
jacorb.orb.objectKeyMap.NameService=file:///home/rnc/NameSingleton.ior
```

This then allows the corbaloc key portion to simply be 'NameService'.

The JacORB utility `dior` may be used to decode IORs. This has an additional command line option to output a corbaloc representation of an IOR. See chapter 22.

### 3.3.3 JacORB Network Event Logging

An enhancement has been added to JacORB that allows a developer to monitor TCP and SSL connections. Note that for both of these implementations full information may only retrieved with a successful connection; e.g. if the connection could not be established there will be no certificates.

#### TCP Monitoring

To monitor TCP connections a developer should implement the following interface

```
package org.jacorb.orb.listener;
public interface TCPConnectionListener extends EventListener
    void connectionOpened(TCPConnectionEvent e);
    void connectionClosed(TCPConnectionEvent e);
```

The classname should then be specified in the property

```
jacorb.net.tcp_listener
```

The standard java event interface is followed; the developer's code will receive the TCPConnectionEvent which allows the following information to be retrieved:

```
    public String getLocalIP()
    public int getLocalPort()
    public String getRemoteIP()
    public int getRemotePort()
```

Note that the TCPConnectionEvent extends java.util.EventObject and the EventObject.getSource operation will return the IIOPConnection of the TCP connection.

### SSL Monitoring

To monitor SSL sessions a developer should implement the following interface

```
package org.jacorb.orb.listener;
public interface SSLSessionListener extends EventListener
    void sessionCreated(SSLSessionEvent e);
    void handshakeException(SSLSessionEvent e);
    void keyException(SSLSessionEvent e);
    void peerUnverifiedException(SSLSessionEvent e);
    void protocolException(SSLSessionEvent e);
    void sslException(SSLSessionEvent e);
```

The classname should then be specified in the property

```
jacorb.security.ssl.ssl_listener
```

The standard java event interface is followed; the developer's code will receive the SSLSessionEvent which allows the following information to be retrieved:

```
    public String getLocalIP()
    public int getLocalPort()
    public String getRemoteIP()
    public int getRemotePort()
    public String getRemoteDN()
    public X509Certificate[] getPeerCertificateChain()
```

Note that getRemoteDN will simply return a concatenated string of the certificates. For that reason it is deprecated; getPeerCertificateChain should be used instead as that allows a developer to extract specific information from the certificate. In order to detect a succesful handshake the implementation delegates to the JSSE `javax.net.ssl.HandShakeCompletedListener`. When using JDK1.3 JSSE the JSSE may not throw for instance a handshakeException but a sslException. Similar to above, SSLSessionEvent extends java.util.EventObject. The EventObject.getSource operation will return the source of the HandshakeCompletedEvent.

### 3.3.4 JacORB IORMutator

An enhancement has been added to JacORB that allows a developer to alter incoming and outgoing objects at a very low level within the ORB. While the majority of the users would not require this ability, it is useful within scenarios where for instance, a user is running with legacy network elements which have multiple, identical IP addresses. This allows them to mutate the IORs as shown below.

**This is a very powerful ability that must be used with caution. As it operates at the CDRStream level it is easy to break the ORB and cause unpredictable behaviour**

#### Adding a Mutator

The developer should firstly extend the following abstract class.

```
package org.jacorb.orb.IORMutator;
public abstract class IORMutator
    protected org.omg.ETF.Connection connection;

    public abstract IOR mutateIncoming (IOR object);
    public abstract IOR mutateOutgoing (IOR object);
```

The classname should then be specified in the property

```
jacorb.iormutator
```

The IORMutator class also has a `org.omg.ETF.Connection connection` variable. This variable will be updated with the current transport information for the respective streams. Note, altering the information within the transport is undefined. The mutateIncoming operation will be called for CDRInputStream operations and the mutateOutgoing for CDROuputStream operations.

### 3.3.5 Using custom socket factories

You may plug in custom socket factories that'll be used by JacORB to create sockets and server sockets. Each factory needs to implement a JacORB specific interface. To make your factory available to JacORB you need to set the appropriate configuration property to the classname of your custom factory. See the following sections for the available factories and their details. Please also see the javadoc documentation of the specified interfaces for the contract your custom factories must adhere to. For convenience JacORB also offers some abstract base classes that pre-implement some functionality and that you may choose to subclass.

#### socket factory

This factory is used by JacORB to create an outgoing non-SSL connection.

**property** jacorb.net.socket_factory

**implemented interface** org.jacorb.orb.factory.SocketFactory

**base class** org.jacorb.orb.factory.AbstractSocketFactory

### server socket factory

This factory is used by JacORB to create a server socket for incoming non-SSL connections.

**property** jacorb.net.server_socket_factory

**implemented interface** org.jacorb.orb.factory.ServerSocketFactory

**base class** org.jacorb.orb.factory.AbstractSocketFactory

### SSL socket factory

This factory is used by JacORB to create an outgoing non-SSL connection.

**property** jacorb.ssl.socket_factory

**implemented interface** org.jacorb.orb.factory.SocketFactory

### SSL server socket factory

This factory is used by JacORB to create a server socket for incoming SSL connections.

**property** jacorb.ssl.server_socket_factory

**implemented interface** org.jacorb.orb.factory.ServerSocketFactory

# 4 Getting Started

Before we explain an example in detail, we 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.

## 4.1 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 (Java interfaces, helper and holder classes, as well as stubs and skeletons).
3. write an implementation for the Java 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 and makes remote invocations, i.e. CORBA calls.

## 4.2 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 `JacORB2_3_1/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 );
    };
  };
};
```

## 4.3 Generating Java classes

Feeding this file into the IDL compiler

```
$ idl -d ./generated 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 `http://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 [**?**]. 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`, and 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 `./generated` 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.

## 4.4 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 exc
    }
}
```

## 4.5 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.

## 4.6 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.
```

### 4.6.1 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();
    }
}
```

# 5 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 (INS) 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 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.

## 5.1 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 [-Djacorb.naming.ior_filename=<filename>] [-DOAPort=port]
[-Djacorb.naming.time_out=<timeout>]
```

You can also start the Java interpreter explicitly by typing

```
$ jaco jacorb.naming.NameServer [-Djacorb.naming.ior_filename=<filename>]
[-DOAPort=port] [-Djacorb.naming.time_out=<timeout>]
```

In the example

```
$ ns -Djacorb.naming.ior_filename=/home/me/public_html/NS_Ref
```

we direct the name server process to write location information (its own object reference) 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, need to be able to access the file through a local or shared file system because the file is read as a 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 is clients and servers are run on a single machine that may 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). In JacORB 1.4, the file name argument was made optional because the JacORB 1.4 name server also answers requests that are made using simplified corbaloc: URLs of the form `corbaloc::ip-address:port/NameService`. This means that all you need to know to construct an object reference to your name service is the IP address of the machine and the port number the server process is listening on (the one specified using `-DOAPort=<port>`).

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 port number on which you want the name service to listen for incoming requests. If this parameter is not set, the name server will come up on the first free port it is provided with by the operating system. The port number can also be set using specific properties in the properties file, but the -DOAPort=¡port¿ switch was added merely for convenience.

The last 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.

## 5.2 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.

## 5.3 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.

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

The graphical front-end to the name service can be started by calling

```
$ nmg
```

The GUI front-end will simply look up the default context and display its contents. Figure 5.1 gives a screen shot.

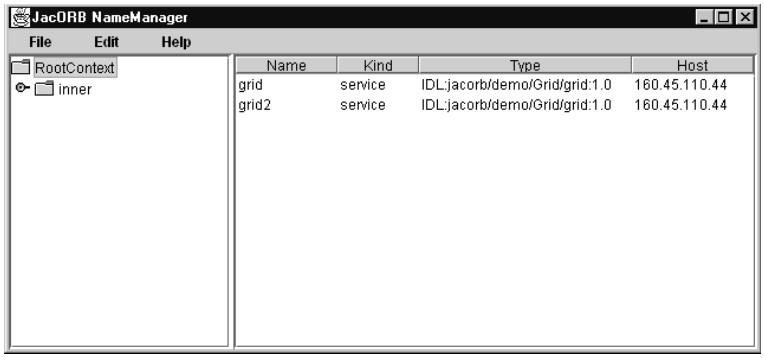


Figure 5.1: NameManager Screenshot

NameManager has menus that let you bind and 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.

# 6 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 [**?**]. Other references that explain the POA are [**?**, **?**]. 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.

## 6.1 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.

## 6.2 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.

# 7 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 [**?**].

## 7.1 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.

## 7.2 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 either notify the ImR manually by using the command `imr_mg setdown AServerName` or allow the ImR to detect the crashed server and restart it if necessary.

## 7.3 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.

## 7.4 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 be 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 11 for more information.

# 8 Dynamic Management of Any Values

*by Jason Courage*

The purpose of this chapter is to describe the DynAny specification, which is the specification for the dynamic management of Any values. This chapter only describes the main features of the DynAny specification; for the complete specification consult the appropriate chapter of the CORBA specification available from the OMG.

## 8.1 Overview

DynAny objects are used to dynamically construct and traverse Any values. A DynAny can represent a value of a basic type, such as boolean or long, or a constructed type, such as enum or struct.

## 8.2 Interfaces

The UML diagram below shows the relationship between the interfaces in the org.omg.DynamicAny module.

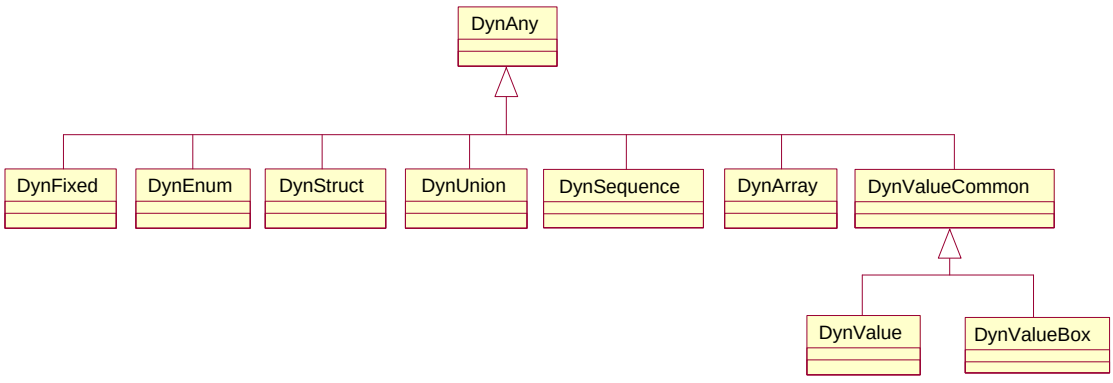


Figure 8.1: DynAny Relationships

The DynAny interface is the base interface that represents values of the basic types. For each constructed type there is a corresponding interface that extends the DynAny interface and defines operations

specific to the constructed type. The table below lists the interfaces in the DynamicAny module and the types they represent.

| Interface | Type |
|---|---|
| DynAny | basic types (boolean, long, etc.) |
| DynFixed | fixed |
| DynEnum | enum |
| DynStruct | struct |
| DynUnion | union |
| DynSequence | sequence |
| DynArray | array |
| DynValue* | non-boxed valuetype |
| DynValueBox* | boxed valuetype |

* Not currently implemented by JacORB.

## 8.3 Usage Constraints

Objects that implement interfaces in the DynamicAny module are intended to be local to the process that constructs and uses them. As a result, references to these objects cannot be exported to other processes or externalized using ORB::object_to_string; an operation that attempts to do so will throw the MARSHAL system exception.

## 8.4 Creating a DynAny Object

The DynAnyFactory interface is used to create a DynAny object. There are two operations for creating a DynAny object; these are listed in the table below.

| Operation | Description |
|---|---|
| create_dyn_any | Constructs a DynAny object from an Any value |
| create_dyn_any_from_type_code | Constructs a DynAny object from a TypeCode |

The example below illustrates how to obtain a reference to the DynAnyFacory object and then use it to construct a DynAny object with each of the create operations. Exception handling is omitted for brevity.

The following line of code imports the classes in the DynamicAny package.

```
import org.omg.DynamicAny.*;
```

The following code segment obtains a reference to the DynAnyFacory object.

```
DynAnyFactory factory = null;
DynAny DynAny = null;
DynAny DynAny2 = null;
org.omg.CORBA.Any any = null;
org.omg.CORBA.TypeCode tc = null;
org.omg.CORBA.Object obj = null;

// obtain a reference to the DynAnyFactory
obj = orb.resolve_initial_references ("DynAnyFactory");

// narrow the reference to the correct type
factory = DynAnyFactoryHelper.narrow (obj);
```

The following code segment creates a DynAny with each of the create operations.

```
// create a DynAny object from an Any
any = orb.create_any ();
any.insert_long (1);
DynAny = factory.create_dyn_any (any);

// create a DynAny object from a TypeCode
tc = orb.get_primitive_tc (org.omg.CORBA.TCKind.tk_long);
DynAny2 = factory.create_dyn_any_from_type_code (tc);
```

If the Any value or TypeCode represents a constructed type then the DynAny can be narrowed to the appropriate subtype, as illustrated below.

The following IDL defines a struct type.

```
// example struct type
struct StructType
{
   long field1;
   string field2;
};
```

The following code segment illustrates the creation of a DynStruct object that represents a value of type StructType.

```
StructType type = null;
DynStruct dynStruct = null;
```

```
// create an Any that contains an object of type StructType
type = new StructType (999, "Hello");
any = orb.create_any ();
StructTypeHelper.insert (any, type);

// construct a DynAny from an Any and narrow it to a DynStruct
dynStruct = (DynStruct) factory.create_dyn_any (any);
```

## 8.5 Accessing the Value of a DynAny Object

The DynAny interface defines a set of operations for accessing the value of a basic type represented by a DynAny object. The operation to get a value of basic type <type> from a DynAny has the form get_<type>. The operation to insert a value of basic type <type> into a DynAny has the form insert_<type>. A TypeMismatch exception is thrown if the type of the operation used to get/insert a value into a DynAny object does not match the type of the DynAny.

The operations for accessing the value of a constructed type represented by a DynAny are defined in the interface specific to the constructed type. For example, the DynStruct interface defines the operation get_members, which returns a sequence of name/value pairs representing the members of the struct or exception represented by a DynStruct object.

## 8.6 Traversing the Value of a DynAny Object

DynAny objects can be viewed as an ordered collection of component DynAnys. For example, in a DynStruct object the ordered collection of component DynAnys is the members of the struct or exception it represents. For DynAny objects representing basic types or constructed types that do not have components, the collection of component DynAnys is empty.

All DynAny objects have a current position. For DynAnys representing constructed types that have components, the current position is the index of the component DynAny that would be obtained by a call to the current_component operation (described in the table below). The component DynAnys of a DynAny object are indexed from 0 to n-1, where n is the number of components. For DynAnys representing basic types, or constructed types that do not have components, the current position is fixed at the value -1.

The operations for traversing the component DynAnys of a DynAny object are common to all DynAny subtypes, hence they are defined in the DynAny base interface. The table below lists the operations available for traversing a DynAny object.

| Operation | Description |
|---|---|
| `seek` | `Sets the current position to the specified index` |

| Operation | Description |
|---|---|
| `rewind` | `Sets the current position to the first component (index 0)` |
| `next` | `Advances the current position to the next component` |
| `component_count` | `Returns the number of components` |
| `current_component` | `Returns the component at the current position` |

The following code segment illustrates one way of traversing the component DynAnys of a DynStruct object. As the DynStruct is traversed, the value of each component is obtained and printed. Exception handling is omitted for brevity.

```
DynAny curComp = null;

// print the value of the first component
curComp = dynStruct.current_component ();
System.out.println ("field1 = " + curComp.get_long ());

// advance to the next component
dynStruct.next ();

// print the value of the second component
curComp = dynStruct.current_component ();
System.out.println ("field2 = " + curComp.get_string ());
```

The next code segment illustrates another way to perform the same task.

```
// go back to the first component
dynStruct.rewind ();  // same as calling seek (0)

// print the value of the first component
System.out.println ("field1 = " + dynStruct.get_long ());

// advance to the next component
dynStruct.seek (1);

// print the value of the second component
System.out.println ("field2 = " + dynStruct.get_string ());
```

As the second code segment illustrates, if the component DynAny represents a basic type, its value can be extracted (or inserted) by calling the accessor operation on the parent DynAny directly, rather than first obtaining the component using the current_component operation.

## 8.7 Constructed Types

This section describes the interfaces in the DynamicAny module that represent the constructed types supported by JacORB. Each of these interfaces extends the DynAny interface.

### 8.7.1 DynFixed

A DynFixed object represents a fixed value. Since IDL does not have a generic type to represent a fixed type, the operations in this interface use the IDL string type. The value represented by a DynFixed object can be accessed (as a string) using the get_value and set_value operations.

A DynFixed object has no components.

### 8.7.2 DynEnum

A DynEnum object represents a single enumerated value. The integer (ordinal) value of the enumerated value can be accessed with the get_as_ulong and set_as_ulong operations. The string (IDL identifier) value of the enumerated value can be accessed with the get_as_string and set_as_string operations.

A DynEnum object has no components.

### 8.7.3 DynStruct

A DynStruct object represents a struct value or an exception value. The current_member_name and current_member_kind operations return the name and TCKind value of the TypeCode of the member at the current position of the DynStruct. The members of the DynStruct can be accessed with the get_members and set_members operations.

The component DynAnys of a DynStruct object are the members of the struct or exception. A DynStruct representing an empty exception has no components.

### 8.7.4 DynUnion

A DynUnion object represents a union value. The value of the discriminator can be accessed using the get_discriminator and set_discriminator operations.

If the discriminator is set to a value that names a member of the union then that member becomes active. Otherwise, if the value of the discriminator does not name a member of the union then there is no active member.

If there is an active member, the member operation returns its value as a DynAny object, and the member_name and member_kind operations return its name and the TCKind value of its TypeCode. These operations throw an InvalidValue exception if the union has no active member.

A DynUnion object can have either one or two components. The first component is always the discriminator value. The second component is the value of the active member, if one exists.

### 8.7.5 DynSequence

A DynSequence object represents a sequence. The length of the sequence can be accessed using the get_length and set_length operations. The elements of the sequence can be accessed using the get_elements and set_elements operations.

The component DynAnys of a DynSequence object are the elements of the sequence.

### 8.7.6 DynArray

A DynArray object represents an array. The elements of the array can be accessed using the get_elements and set_elements operations.

The component DynAnys of a DynArray object are the elements of the array.

## 8.8 Converting between Any and DynAny Objects

The DynAny interface defines operations for converting between Any objects and DynAny objects. The from_any operation initialises the value of a DynAny with the value of a specified Any. A TypeMismatch exception is thrown if the type of the Any does not match the type of the DynAny. The to_any operation creates an Any from a DynAny.

As an example of how these operations might be useful, suppose one wants to dynamically modify the contents of some constructed type, such as a struct, which is represented as an Any. The following steps will accomplish this task:

1. A DynStruct object is constructed from the TypeCode of the struct using the DynAnyFactory::create_dyn_any_from_type_code operation.
2. The DynAny::from_any operation is used to initialise the value of the DynStruct with the value of the Any.
3. The contents of the DynStruct can now be traversed and modified.
4. A new Any can be created to represent the modified struct using the DynAny::to_any operation.

## 8.9 Further Examples

The demo/dynany directory of the JacORB repository contains example code illustrating the use of DynAny objects. Further code can be found in the org.jacorb.test.orb.dynany package of the JacORB-Test repository.

# 9 Objects By Value

Until CORBA 2.3, objects could only be passed using reference semantics: there was no way to specify that object state should be copied along with an object reference. A further restriction of the earlier CORBA versions was that all non-object types (structs, unions, sequences, etc.) were *values*, so you could not use, e.g. a reference-to-struct to construct a graph of structure values that contained shared nodes. Finally, there was no inheritance between structs.

All these shortcomings are addressed by the *objects-by-value* (OBV) chapters of the CORBA specification: the addition of stateful value types supports copy semantics for objects and inheritance for structs, boxed value types introduce reference semantics for base types, and abstract interfaces determine whether an argument is sent by-value or by-reference by the argument's runtime type. The introduction of OBV into CORBA presented a major shift in the CORBA philosophy, which had been to strictly avoid any dependence on implementation details (state, in particular). It also added a considerable amount of marshaling complexity and interoperability problems. (As a personal note: Even in CORBA 2.6, the OBV marshaling sections are still not particularly precise...)

JacORB 2.0 implements most of the OBV specification. Boxed value types and regular value types work as prescribed in the standard (including value type inheritance, recursive value types, and factories). Still missing in the current implementation is run-time support for abstract value types (although the compiler does accept the corresponding IDL syntax), and the marshaling of truncatable value types does not yet meet all the standard's requirements (and should thus be called "beta").

## 9.1 Example

To illustrate the use of various kinds of value types, here's an example which is also part of the demo programs in the JacORB distribution. The demo shows the use of boxed value types and a recursive stateful value type. Here's the IDL definition from `demo/value/server.idl`:

```
module demo {
  module value {

     valuetype boxedLong   long;
     valuetype boxedString string;

     valuetype Node {
        public long id;
        public Node next;
     };
```

```
     interface ValueServer {
        string receive_long   (in boxedLong p1, in boxedLong p2);
        string receive_string (in boxedString s1, in boxedString s2);
        string receive_list   (in Node node);
     };
  };
};
```

From the definition of the boxed value type `boxedLong` and `boxedString`, the IDL generates the following Java class, which is simply a holder for the long value. No mapped class is generated for the boxed string value type.

```
package demo.value;

public class boxedLong
    implements org.omg.CORBA.portable.ValueBase
{
    public int value;
    private static String[] _ids = { boxedLongHelper.id() };

    public boxedLong(int initial )
    {
        value = initial;
    }
    public String[] _truncatable_ids()
    {
        return _ids;
    }
}
```

The boxed value definitions in IDL above permit uses of non-object types that are not possible with IDL primitive types. In particular, it is possible to pass Java `null` references where a value of a boxed value type is expected. For example, we can call the operation `receive_long` and pass one initialized `boxedLong` value and a null reference, as show in the following snippet from the client code:

```
    ValueServer s = ValueServerHelper.narrow( obj );
    boxedLong boxL = new boxedLong (774);

    System.out.println ("Passing two integers: "
                        + s.receive_long ( boxL , null ));
```

With a regular `long` parameter, a `null` reference would have resulted in a `BAD_PARAM` exception. With boxed value types, this usage is entirely legal and the result string returned from the `ValueServer` object is `''one or two null values''`.

A second new possibility of the reference semantics that can be achieved by "boxing" primitive IDL types is *sharing* of values. With primitive values, two variables can have copies of the same value, but they cannot both refer to the same value. This means that when one of the variables is changed, the other one retains its orignal value. With shared values that are *referenced*, both variables would always point to the same value.

The stateful value type `Node` is implemented by the programmer in a class `NodeImpl` (see the JacORB distribution for the actual code). The relationship between this implementation class and the corresponding IDL definition is not entirely trivial, and we will discuss it in detail below.

## 9.2 Factories

When an instance of a (regular) value type is marshaled over the wire and arrives at a server, a class that implements this value type must be found, so that a Java object can be created to hold the state information. For interface types, which are only passed by reference, something similar is accomplished by the POA, which accepts remote calls to the interface and delivers them to a local implementation class (the *servant*). For value type instances, there is no such thing as a POA, because they cannot be called remotely. Thus, the ORB needs a different mechanism to know which Java implementation class corresponds to a given IDL value type.

The CORBA standard introduces *value factories* to achieve this. Getting your value factories right can be anywhere from trivial to tricky (we will cover the details in a minute), and so the standard suggests that ORBs also provide convenience mechanisms to relieve programmers from writing value factories if possible. JacORB's convenience mechanism is straightforward:

> *If the implementation class for an IDL value type A is named AImpl, resides in the same package as A, and has a no-argument constructor, then no value factory is needed for that type.*

In other words, if your implementation class follows the common naming convention ("...Impl"), and it provides a no-arg constructor so that the ORB can instantiate it, then the ORB has all that it needs to (a) find the implementation class, and (b) create an instance of it (which is then initialized with the unmarshaled state from the wire).

This mechanism ought to save you from having to write a value factory 99% of the time. It works for all kinds of regular value types, including those with inheritance, and recursive types (where a type has members of its own type).

If you do need more control over the instance creation process, or the unmarshaling from the wire, you can write your own value factory class and register it with the ORB using `ORB.register_value_factory(`*repository_id, factory*`)`. The *factory* object needs to implement the interface `org.omg.CORBA.portable.ValueFactory`, which requires a single method:

```
public Serializable read_value (InputStream is);
```

When an instance of type *repository_id* arrives over the wire, the ORB calls the `read_value()` method, which must unmarshal the data from the input stream, create an instance of the appropriate implementation class from it, and return that.

The easiest way to implement this method is to create an instance of the implementation class, and pass it to the `read_value()` method of the given InputStream:

```
public Serializable read_value (InputStream is) {
  A result = new AImpl();
  return is.read_value(result);
}
```

The `InputStream.read_value()` method registers the newly created instance in the stream's indirection table, and then reads the data from the stream and initializes the given `value` instance from it.

The value factory must be registered with the ORB using `register_value_factory()`. As a special convenience (defined in the CORBA standard), if the value factory class for type `A` is called `ADefaultFactory`, then the ORB will find it automatically and use it, unless a different factory has been explicitly registered.

It sometimes causes confusion that you can also define *factory methods* in a value type's IDL. These factory methods are completely unrelated to the unmarshaling mechanism discussed above; they are simply a portable means to declare what kinds of "constructors" a value type implementation should have. They are purely for local use, but since they are "factories", the corresponding methods must also be implemented in the type's `ValueFactory` implementation.

# 10 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.

## 10.1 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 10.1 illustrates this hierarchy.

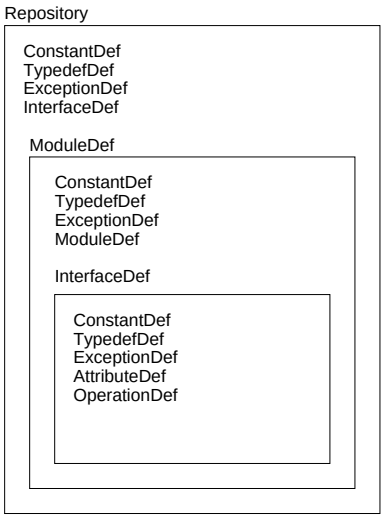


Figure 10.1: Containers in the Interface Repository

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.

## 10.2 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 10.2 illustrates this functionality, where $f^{-1}$ denotes the reverse mapping, or the inverse of the language mapping.

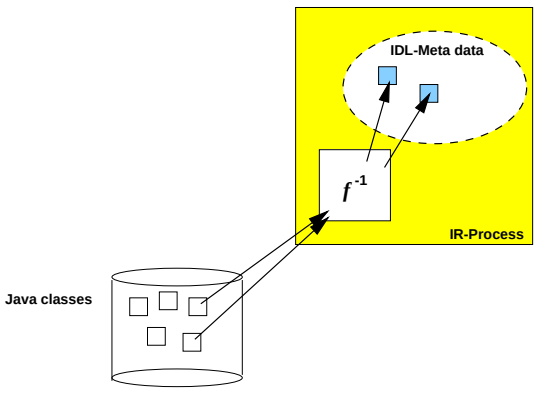


Figure 10.2: The JacORB Interface Repository

## 10.3 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. See below for instructions on generating classes with IR information. 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 [ -i <IOR-string> | -f <filename>]
```

e.g.

```
$ irbrowser
```

Note that if no arguments are supplied it will default to using resolve_initial_references.

Figure 10.3 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.

**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., OperationDef 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.

## 10.4 Interaction between #pragma prefix and -i2jpackage

Generally any use of *#pragma prefix* or *-i2jpackage* should be avoided. if you intend to use a IDL file with the Interface Repository. If there is no other option there is a property that allows you to circumvent that restriction in some cases. Note however that this is a non-standard extension.

If, for example you have the following IDL file:

```
#pragma prefix "org.jacorb.test"

module ir
{
    typedef string StringAlias;
```

```
        typedef sequence<StringAlias> StringAliasList;

        struct TestStruct
        {
            StringAliasList stringList;
        };
    };
```

As you want your generated java files to reside in the package *org.jacorb.test.ir* you need to add *-i2jpackage* as an argument to the *idl* command. `$ idl -ir -i2jpackage ir:org.jacorb.test.ir myIdlFile.idl` Now the generated files are in the directory org/jacorb/test/ir.

As the IR starts it reads in the generated classes and implicitely creates their Repository ID's solely based on the directory structure. e.g. the struct TestStruct will get the Repository ID `IDL:org/jacorb/test/ir/TestStruct:1.0` however the correct Repository ID is `IDL:org.jacorb.test/ir/TestStruct:1.0`.

This will make it impossible for you to lookup the correct Repository ID successfully. starting of the IR will fail if the IR itself needs to look up a Repository ID during start.

As a workaround you can specify the property *jacorb.ir.patch_pragma_prefix=on* to the IR server. this property will cause the IR to change the first component of a requested Repository ID (Repository ID's consists of multiple components delimited with '/' so its org.jacorb.test in this case). If the first component looks like a pragma prefix (contains multiple '.') the '.' will be changed to '/'.

So the incoming request for `IDL:org.jacorb.test/ir/TestStruct:1.0` will be changed to a request for `IDL:org/jacorb/test/ir/TestStruct:1.0` so that the IR will be able to resolve that.

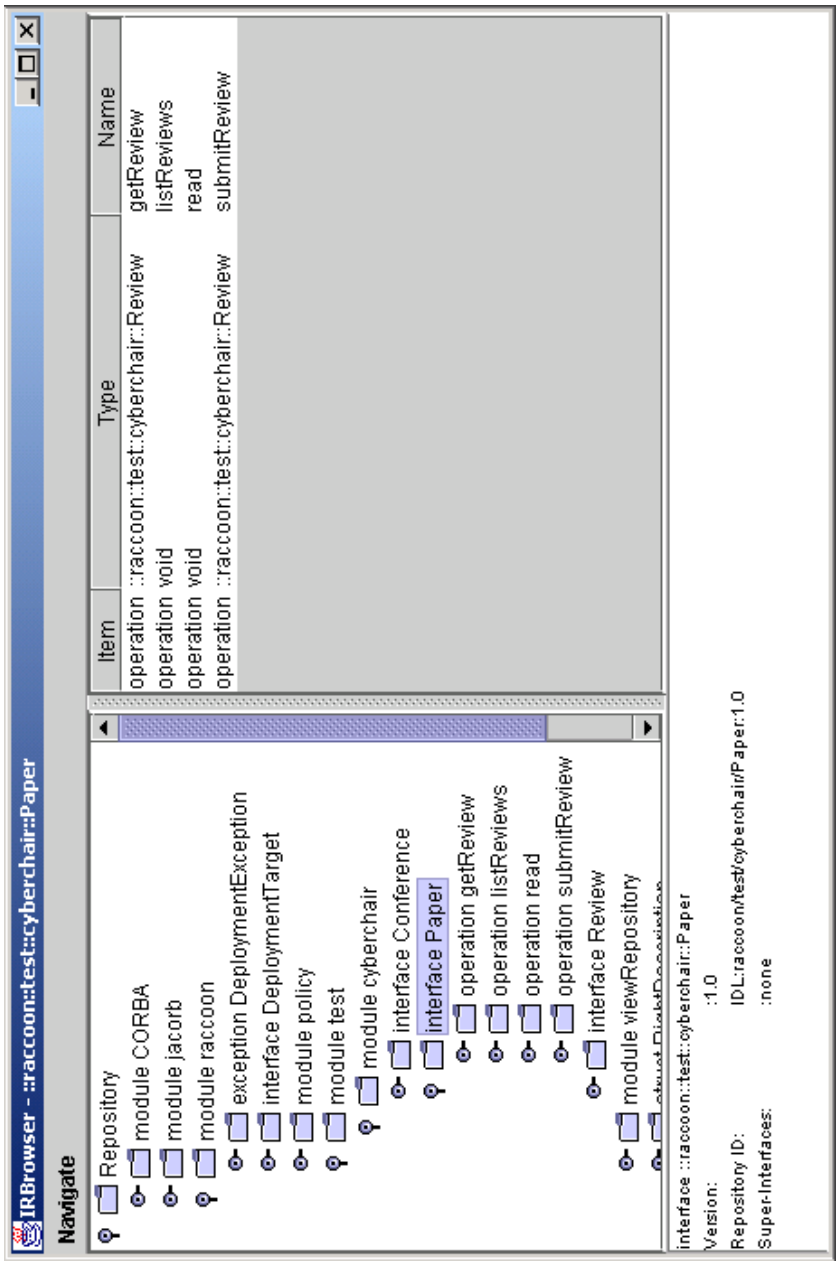


Figure 10.3: IRBrowser Screenshot

# 11 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 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.

**Note:** unlike previous versions of JacORB, as the minimum JDK is 1.4, SSL is enabled by default.

## 11.1 Key stores

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.

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*.

The following section explain how to create key stores for Sun JSSE.

### 11.1.1 Setting up a JSSE key store

To set up key stores with JSSE you can use Java's `keytool`. In order to generate a simple public key infrastructure you can perform the following steps:

1. Create a new key store containing a new public/private key pair with `keytool`. The public key will be wrapped into a self-signed certificate.
2. Export the self-signed certificate from the key store into a file.
3. Import the self-signed certificate into a trust store (or configure that trustees shall be read from key store, see below).

To create a new key store containing a new public/private key pair type:

```
keytool -genkey -alias <alias> -keystore <keystore>
```

If you don't give a key store name `keytool` will create a key store with the name `.keystore` in the user's home directory. The command given above will ask for the following input:

```
Enter keystore password:  changeit
What is your first and last name?
  [Unknown]:  Developer
What is the name of your organizational unit?
  [Unknown]:  cs
What is the name of your organization?
  [Unknown]:  PrismTech
What is the name of your City or Locality?
  [Unknown]:  Berlin
What is the name of your State or Province?
  [Unknown]:  Berlin
What is the two-letter country code for this unit?
  [Unknown]:  Germany
Is CN=Developer, OU=cs, O=PrismTech, L=Berlin, ST=Berlin,
   C=Germany correct?
  [no]:  yes

Enter key password for <testkey>
     (RETURN if same as keystore password):  testkey
```

You can view the entries of the newly created keystore by typing:

```
keytool -keystore <keystore> -list -storepass <password>
```

The output will read for example like this:

```
Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

testkey, Dec 1, 2004, keyEntry,
Certificate fingerprint (MD5): C4:9B:11:97:FF:CD:4C:C9:B3:02:BB:
9A:46:D8:C3:11
```

Now you have a public key certificate that you can present for authentication. The public key contained in the key store is wrapped into a self-signed certificate. This self-signed certificate has to be added to the Java trust store. To do this export the certificate from the key store and import it into the Java trust store located in `<java_home>/jre/lib/security/cacerts`.

To export the self-signed certificate into a file type:

```
keytool -export -keystore <keystore> -alias <alias> -file <filename>
```

To import the certificate into the trust store type:

```
keytool -import -keystore <truststore> -alias <alias> -file <filename>
```

More documentation on key stores can be found in the Java tool documentation for the `keytool` command. Note that 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.

### 11.1.2 Step–By–Step certificate creation

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

1. Create new key stores (File/new) and keypairs (Keys/new) for the CA and for the user.
2. Open the user keys tore (File/open), select the key entry and export the self-signed certificate (Certificates/Export).
3. Open the CA key store 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 key store (Trustees/Delete).
6. Open the user key store 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 key store can be deployed without it. Please note that a failed verification might result in a SignatureException.

## 11.2 Configuring SSL properties

When the ORB is initialized by the application, a couple of properties are read from files and the command line. To turn on SSL support, you have to set the following property to "on":

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

This will just load the SSL classes on startup. The configuration of the various aspects of SSL is done via additional properties.

Configure which SSL socket factory and SSL server socket factory shall be used with the properties:

```
jacorb.ssl.socket_factory=qualified classname
jacorb.ssl.server_socket_factory=qualified classname
```

If you want to use JSSE, then configure the following as qualified classname of SSL Socket Factory and SSL server socket factory:

```
org.jacorb.security.ssl.sun_jsse.SSLSocketFactory
org.jacorb.security.ssl.sun_jsse.SSLServerSocketFactory
```

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

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

The key store file name can either be an absolute path or relative to the home directory. Key stores are searched in this order, and the first one found is taken. If this property is not set, the user will be prompted to enter a key store location on ORB startup.

The password for the key store file can be specified by using the property `jacorb.security.keystore_password`.

```
jacorb.security.keystore_password=secret
```

By default the KeyStore type uses JKS; to change this alter the property `jacorb.security.keystore_type`.

The SSL socket factory algorithms are initialised by default to SunX509. On other JDK implementations (e.g. IBM) this can be changed (to e.g. `IbmX509`) by altering the following properties:

```
jacorb.security.jsse.server.key_manager_algorithm=SunX509
jacorb.security.jsse.server.trust_manager_algorithm=SunX509
jacorb.security.jsse.client.key_manager_algorithm=SunX509
jacorb.security.jsse.client.trust_manager_algorithm=SunX509
```

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 key store
jacorb.security.default_user=brose
jacorb.security.default_password=jacorb
```

Note that when using Sun JSSE: The `javax.net.ssl.trustStore[Password]` properties doesn't seem to take effect, so you may want to add trusted certificates to "normal" key stores. In this case configure JacORB to read certificates from the key store rather than from a dedicated trust store, please set the property

```
jacorb.security.jsse.trustees_from_ks=on
```

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

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

jacorb.security.ssl.server.supported_options=0
jacorb.security.ssl.server.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;
```

### 11.2.1 Protocols

The JSSE is capable of supporting SSL versions 2.0 and 3.0 and Transport Layer Security (TLS) 1.0. To enable different protocols in the JSSE layer use the below properties.

```
jacorb.security.ssl.client.protocols
jacorb.security.ssl.server.protocols
```

Refer to the JSSE documentation for valid SSLSocket/SSLContext protocol values.

### 11.2.2 Client side and server side configuration

On both the client side and the server side supported and required options can be configured. The following tables explain the settings for supported and required options for client and server.

Table 11.1: Client side supported options

| Property with value | Description |
|---|---|
| `jacorb.security.ssl.client.supported_options=20 // EstablishTrustInTarget` | This value indicates that the client can use SSL. Actually, this is default SSL behaviour and must always be supported by the client. |
| `jacorb.security.ssl.client.supported_options=40 // EstablishTrustInClient` | This makes the client load it's own key/certificate from it's key store, to enable it to authenticate to the server. |

Table 11.2: Client side required options

| Property with value | Description |
|---|---|
| `jacorb.security.ssl.client.required_options=20 // EstablishTrustInTarget` | This enforces SSL to be used. |
| `jacorb.security.ssl.client.required_options=40 // EstablishTrustInClient` | This enforces SSL to be used. Actually, this is no meaningfuly value, since in SSL, the client can't force it's own authentication to the server. |

Table 11.3: Server side supported options

| Property with value | Description |
|---|---|
| `jacorb.security.ssl.server.supported_options=1 // NoProtection` | This tells the clients that the server also supports unprotected connections. If NoProtection is set, no required options should be set as well, because they override this value. |
| `jacorb.security.ssl.server.supported_options=20 // EstablishTrustInTarget` | This value indicates that the server supports SSL. Actually, this is default SSL behaviour and must always be supported by the server. This also makes the server load it's key/certificate from the key store. |
| `jacorb.security.ssl.server.supported_options=40 // EstablishTrustInClient` | This value is ignored, because authenticating the client is either required, or not done at all (the client can't force its own authentication). |

Table 11.4: Server side required options

| **Property with value** | **Description** |
|---|---|
| `jacorb.security.ssl.server.required_options=20 // EstablishTrustInTarget` | This enforces SSL to be used. |
| `jacorb.security.ssl.server.required_options=40 // EstablishTrustInClient` | This enforces SSL to be used, and will request the client to authenticate. It also will load trusted certificates for the authentication process. |

## 11.3 SecureRandom Plugin System

Under certain platforms (e.g. J2ME CDC platforms) when the JSSE initializes its random number generator it may spawn a large number of threads and/or have a significant start-up time. This overhead may be unacceptable.

In order to allow developers to provide their own initialization routines for SecureRandom a plugin class may be provided. A developer should implement the following interface.

```
package org.jacorb.security.ssl.sun_jsse;

public interface JSRandom
{
    SecureRandom getSecureRandom ();
}
```

The classname should then be specified in the property

```
jacorb.security.randomClassPlugin
```

which will be instantiated at runtime. If this property has been specified the SSLSocket factories will call `getSecureRandom` to pass through to the SSLContext. Otherwise, the JSSE will use its default values.

Two example implementations; `JSRandomImpl` and `JSRandomImplThread` are provided. `JSRandomImpl` explicitly initializes a SecureRandom with a fixed seed value. Note that the seed is a hardcoded value (4711). As using such a seed is a security risk it is not recommended that this code be used in a production system. The second, using initSecureRandom (see below)

```
public class JSRandomImplThread implements JSRandom {

    public static void initSecureRandom() { ... }
}
```

allows the developer to initialize a single static SecureRandom in a separate thread at the start of their main before any ORB calls are done.

## 11.4 Security and corbaloc

If you want to put together a corbaloc that points to your SSL enabled server object the following needs to be ensured:

normally an IOR string contains additional components that describe the exact SSL setup for a given server object. However this additional information cannot be attached to a corbaloc. JacORB provides an extensions to address that shortcoming.

By using the JacORB specific protocol extension *ssliop* you can tell ORB that the corbaloc points to a SSL enabled target. When the corbaloc is resolved with orb.string_to_object() and the protocol extension is set, JacORB will act as the target had the SSL specific tagged components set[1].

By default the SSL option EstablishTrustInTarget will be used both for supported and required SSL options of the created stub. Using the property jacorb.security.ssl.corbaloc_ssliop.supported_options this can be further customized. Have a look at the configuration chapter for more details.

Example of a corbaloc

```
corbaloc:ssliop:1.2@hostname:port/object_key
```

[1] you need to ensure that the corbaloc used GIOP v1.2 as otherwise tagged components are not supported

# 12 MIOP

JacORB has an implementation of MIOP written as an ETF plugin. This conforms to version 03-01-11 of the Unreliable Multicast Inter-ORB Protocol specification.

## 12.1 Enabling the MIOP Transport

In order to enable the ETF transport plugin the following configuration properties must be altered.

```
jacorb.transport.factories
jacorb.transport.client.selector
```

By default these properties are configured to use the IIOP transport. For example to select both IIOP and MIOP transports:

```
jacorb.transport.factories=org.jacorb.orb.iiop.IIOPFactories,
                           org.jacorb.orb.miop.MIOPFactories
jacorb.transport.client.selector=
                           org.jacorb.orb.miop.MIOPProfileSelector
```

## 12.2 Configuring the MIOP Transport

A number of extra configuration properties have been added for the transport.

Table 12.1: MIOP Configuration

| Property | Description | Type |
|---|---|---|
| `jacorb.miop.timeout` | Timeout used in MIOP requests. Default is 100. | integer |
| `jacorb.miop.time_to_live` | TTL used for multicast UDP packets. Default is 5 seconds. | integer |
| `jacorb.miop.incomplete_messages_threshold` | Maximum number of incomplete messages allowed. Default 5. | integer |
| `jacorb.miop.message_completion_timeout` | Timeout for packet collection to be completed. Default 500ms. | integer |

Table 12.1: MIOP Configuration

| Property | Description | Type |
|---|---|---|
| jacorb.miop.packet_max_size | This is the maximum size of the frame buffer. This defaults to 1500 bytes which is the typical value for most network interfaces. From this the IP, UDP and UMIOP headers will be deducted which will leave 1412 bytes for the MIOP packet. | integer |

## 12.3 MIOP Example

A new demo has been included within <JacORB>/demo/miop. This section will describe how to run this demo including its use of MIOP corbaloc strings.

Assuming the developer has installed Ant version 1.7.1 or above then the example may be compiled by typing ant within the <JacORB>/demo/miop directory. The classes will be compiled to <JacORB>/classes which may need to be added to the classpath.

To run the server:

```
jaco demo.miop.Server
```

To run the client:

```
jaco demo.miop.Client
```

This is the simplest configuration and will simply send two oneway requests via UDP to the server. By default the Server will write out a miop.ior file containing the following corbaloc:

```
corbaloc:miop:1.0@1.0-TestDomain-1/224.1.239.2:1234;
         iiop:1.2@10.1.0.4:38148/4222541922/%00%16%0F%205=%25%02%01I%0C
```

The Group IIOP Profile key string will not remain constant. The server takes a single optional argument:

-noGroupProfile Don't write IIOP Group Profile request.

This will create a corbaloc as shown below and is useful for interoperating with ORBs that do not support the Group Profile.

```
corbaloc:miop:1.0@1.0-TestDomain-1/224.1.239.2:1234
```

The Client takes two optional arguments:

| | |
|---|---|
| -fragment | Trigger fragmentation by sending a larger request. |
| [IOR—Corbaloc] | Don't use miop.ior but this supplied IOR or Corbaloc. |

The second optional argument is useful if interoperating with another ORB.

### 12.3.1 Two way requests and MIOP

The demo client does an unchecked_narrow on the supplied corbaloc/URL. This is because a MIOP URL does not normally support a two-way is_a request unless a Group IIOP profile has also been encoded into the corbaloc. By default the JacORB demo server will create the Group IIOP profile as well:

```
corbaloc:miop:1.0@1.0-TestDomain-1/224.1.239.2:1234;
         iiop:1.0@10.1.0.4:36840/7150661784/%00%16%0F%1B*@2%02,%1A
```

It is not guaranteed that other ORBs (e.g. TAO) will create the Group IIOP profile.

# 13 BiDirectional GIOP

BiDirectional GIOP has its main use in configurations involving callbacks with applets or firewalls where it sometimes isn't possible to open a direct connection to the desired target. As a small example, imagine that you want to monitor the activities of a server via an applet. This would normally be done via a callback object that the applet registers at the server, so the applet doesn't have to poll the server for events. To accomplish this without BiDirectional GIOP, the server would have to open a new connection to the client which will not work because applets usually arent allowed to act as servers, i.e. open ServerSockets. At this point BiDirectional GIOP can help because it allows to reuse the connection the applet opened to the server for GIOP requests from the server to the applet (which isn't allowed in "standard" GIOP).

## 13.1 Setting up Bidirectional GIOP

Setting up BiDirectional GIOP consists of two steps:

1. Setting an ORBInitializer property and creating the BiDir policy
2. Adding this policy to the servant's POA.

### 13.1.1 Setting the ORBInitializer property

The first thing that is necessary for BiDirectional GIOP to be available is the presence of the following property, which can be added by the usual ways (see chapter 3):

```
org.omg.PortableInterceptor.ORBInitializerClass.bidir_init=
    org.jacorb.orb.giop.BiDirConnectionInitializer
```

If this property is present on ORB startup, the corresponding policy factory and interceptors will be loaded.

### 13.1.2 Creating the BiDir Policy

Creating the necessary BiDir Policy is done via a policy factory hidden in the ORB.

```
import org.omg.BiDirPolicy.*;
```

```
import org.omg.CORBA.*;

[...]

Any any = orb.create_any();
BidirectionalPolicyValueHelper.insert( any, BOTH.value );

Policy p  = orb.create_policy( BIDIRECTIONAL_POLICY_TYPE.value,
                               any );
```

The value of the new policy is passed to the factory inside of an any. The ORB is the told to create a policy of the specified type with the specified value. The newly created policy is then used to create a user POA. Please note that if *any* POA of has this policy set, *all* connections will be enabled for BiDirectional GIOP, that is even those targeted at object of POAs that don't have this policy set. For the full source code, please have a look at the bidir demo in the `demo` directory.

## 13.2 Verifying that BiDirectional GIOP is used

From inside of your application, it is impossible to tell whether requests arrived over a unidirectional or BiDirectional connection. Therefore, to check if connections are used in both directions, you can either use a network monitoring tool or take a look at JacORBs output to tell you if your server created a new connection to the client, or if the existing one is being reused.

If the debug level is set to 2 or larger, the following output on the server side will tell you that a connection is being reused:

```
[ ConnectionManager: found conn to target <my IP>:<my port> ]
```

If, on the other hand, the connection is not being reused, the client will show the following output:

```
[ Opened new server-side TCP/IP transport to <my host>:<my port> ]
```

## 13.3 TAO interoperability

There is one problem that may prevent TAO and JacORB to interoperate using BiDirectional GIOP: If JacORB uses IP addresses as host names (JacORBs default) and TAO uses DNS names as host names (TAOs default), connections from JacORB clients to TAO servers will not be reused. If, on the other hand, both use the same "format" for host addresses, interoperability will be successful. There are two ways to solve this problem:

1. Use `''-ORBdotteddecimaladdresses 1''` as an command line argument to the TAO server.
2. Recompile JacORB with DNS support (See the INSTALL file for more information).

# 14 Portable Interceptors

Since revision 1.1 JacORB provides support for Portable Interceptors These interceptors are compliant to the standard CORBA specification. 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>
```

For compatibility reasons with the spec, the properties format may also be like this:

```
org.omg.PortableInterceptor.ORBInitializerClass.<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 $\geq 2$ 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 `0x4A414301` 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.

## 14.1 Interceptor ForwardRequest Exceptions

Several of the interceptor types may throw a ForwardException such as `ClientRequestInterceptor send_request`. A developer may wish to do this if, for instance, a new policy is being applied to the object to switch to a SSL connection type as suggested within section **??**.

A current limitation of the specification (CORBA 3; 02-06-33) is that it is impossible to detect whether the call has previously been thrown for the same client request. Thus it is possible to enter an infinite loop throwing ForwardRequest at this point. This issue was first submitted to the OMG in May 2002 under number 5266.

In order to allow developers more flexibility when writing their interceptors PrismTech have enhanced the exception handling as follows. We have chosen one of the solutions proposed within issue 5266; namely to allow forward_reference() to be accessed in send_request() as well as in receive_other(). i.e. returning the object from the previous ForwardRequest if that has been thrown and null otherwise.

A typical use of this might be

```
public void send_request( ClientRequestInfo ri )
{
   if (ri.effective_profile().tag == TAG_INTERNET_IOP.value &&
       ri.forward_reference() == null)
   {
      // Do some processing, throw a forward request.
   }
}
```

This allows the developer to conditionally throw a forward request while using forward_reference() to prevent infinite loops.

# 15 Asynchronous Method Invocation

JacORB allows you to invoke objects asynchronously, as defined in the *Messaging* chapter of the CORBA specification (chapter 22 in CORBA 3.0). Only the callback model is implemented at this time; there is no support for polling yet.

Asynchronous Method Invocation (AMI) means that when you invoke a method on an object, control returns to the caller immediately; it does not block until the reply has been received from the remote object. The results of the invocation are delivered later, as soon as they are received by the client ORB. Asynchronous Invocation is entirely a client-side feature. The server is never aware whether it is invoked synchronously or asynchronously.

In the callback model, replies are delivered to a special *ReplyHandler* object that is registered at the client side when the asynchronous invocation is started. Here is a brief example for this (see the *Messaging* specification for further details). Suppose you have a Server object, defined in a file server.idl.

```
interface Server
{
    long operation (in long p1, inout long p2);
};
```

The first step is to compile this IDL definition with the "ami_callback" compiler switch:

```
idl -ami_callback server.idl
```

This lets the compiler generate an additional ReplyHandler class, named AMI_ServerHandler. For each operation of the Server interface, this class has an operation with the same name that receives the return value and out parameters of the original operation. There is an additional method named operation_excep that is called if the invocation raises an exception. If it were defined in IDL, the ReplyHandler class for the above Server would look like this:

```
interface AMI_ServerHandler : Messaging::ReplyHandler
{
   void operation (in long ami_return_val, in long p2);
   void operation_excep (in Messaging::ExceptionHolder excep_holder);
};
```

To implement this interface, extend the corresponding POA class (or use the tie approach), as with any CORBA object:

```
public class AMI_ServerHandlerImpl extends AMI_ServerHandlerPOA
{
     public void operation (int ami_return_val, int p2)
     {
         System.out.println ("operation reply received");
     }

     public void operation_excep
              (org.omg.Messaging.ExceptionHolder excep_holder)
     {
         System.out.println ("received an exception");
     }

}
```

For each method $m$ of the original Server interface, the IDL compiler generates a special method sendc_$m$ into the stub class if the "ami_callback" switch is on. The parameters of this method are (1) a reference to a ReplyHandler object, and (2) all *in* or *inout* parameters of the original operation, with their mode changed to *in* (*out* parameters are omitted from this operation). The sendc operation does not have a return value.

To actually make an asynchronous invocation, an instance of the ReplyHandler needs to be created, registered with the ORB, and passed to the sendc method. The code for this might look as follows:

```
ORB    orb = ...
Server s   = ...

// create handler and obtain a CORBA reference to it
AMI_ServerHandler h = new AMI_ServerHandlerImpl()._this (orb);

// invoke sendc
((_ServerStub)s).sendc_operation (h, 4, 5);
```

Note that the sendc operation is only defined in the stub, and therefore the cast is necessary to invoke it. There is not yet any consensus in the OMG whether the sendc operation should also be declared in any of the Java interfaces that make up the Server type. Thus, the fact that you need to make a cast to the stub class may change in a future version of JacORB.

If you want to try asynchronous invocations with code such as above, make sure that your client process does something else or at least waits after the invocation has been made, otherwise it will likely exit before the reply can be delivered to the handler.

The *Messaging* specification also defines a number of CORBA policies that allow you to control the timing of asynchronous invocations. Since these policies are applicable to both synchronous and asynchronous invocations, we describe them in a separate section (see chapter 16).

# 16 Quality of Service

JacORB implements a subset of the QoS policies defined in chapter 22.2 of the CORBA 3.0 specification. In the following, we describe each of the policies we have currently implemented, along with notes on particular JacORB issues concerning each policy. Policies not listed in the following are not yet implemented.

As of yet, all policies described in this chapter are *client-side override policies*. The CORBA specification uses the term for any policy that is explicitly set and thus overrides system defaults. Policies can be set at different scopes: per object, per thread, or per ORB. The current JacORB implementation only supports object and ORB scopes. In general, the following steps are necessary:

**Step 1.** Get an `any` from the ORB and put the value for the policy into it.

**Step 2.** Get a Policy object from the ORB which encapsulates the desired value (the `any` value from the previous step).

**Step 3.** Apply the policy to a particular object using the `_set_policy_override()` operation on the object reference.

**Step 3.** alternatively: set the policy ORB-wide using the `set_policy_overrides()` operation on the ORB's `PolicyManager` object.

Below is the code that corresponds to the steps listed above, using the *SyncScopePolicy* (described in the following section) as an example. Also, have a look at the demo program in `demo/policies`:

```
SomeCorbaType     server = ...
org.omg.CORBA.ORB orb    = ...
org.omg.CORBA.Any a      = orb.create_any();
a.insert_short(SYNC_WITH_SERVER.value); // the value for that policy
try
{
    Policy p = orb.create_policy(SYNC_SCOPE_POLICY_TYPE.value, a);
    server._set_policy_override (new Policy[]{ p },
                                 SetOverrideType.ADD_OVERRIDE);

    // get the ORB's policy manager
    PolicyManager policyManager =
        PolicyManagerHelper.narrow(
            orb.resolve_initial_references("ORBPolicyManager"));
```

```
    // set an ORB-wide policy
    policyManager.set_policy_overrides( new Policy[]{ p },
                                        SetOverrideType.ADD_OVERRIDE);
}
catch (PolicyError e)
{
    throw new RuntimeException ("policy error: " + e);
}
```

The above is portable code that relies only on standardized CORBA APIs to create and set policies. Because this code is somewhat cumbersome to write, JacORB also allows you to simplify it by creating the Policy object directly via its constructor, as shown below. Note that this is non-portable code:

```
SomeCorbaType server  = ...

Policy p = new org.jacorb.orb.policies.SyncScopePolicy
                                    (SYNC_WITH_TARGET.value);
server._set_policy_override (new Policy[]{ p },
                             SetOverrideType.ADD_OVERRIDE);
```

See the package org.jacorb.orb.policies to find out which constructors are defined for the individual policy types.

## 16.1 Sync Scope

The *SyncScopePolicy* specifies at which point a oneway invocation returns to the caller. (The policy is ignored for non-oneway invocations.) There are four possible values:

**SYNC_NONE** The invocation returns immediately.

**SYNC_WITH_TRANSPORT** The invocation returns after the request has been passed to the transport layer.

**SYNC_WITH_SERVER** The server sends an acknowledgement back to the client when it has received the request, but *before* actually invoking the target. The client-side call blocks until this acknowledgement has been received.

**SYNC_WITH_TARGET** An ordinary reply is sent back by the server, *after* the target invocation has completed. The client-side call blocks until this reply has been received.

The default mechanism in JacORB is *SYNC_WITH_TRANSPORT*, since the call to the socket layer is a synchronous one. In order to implement *SYNC_NONE*, an additional thread is created on the fly which in turn calls the socket layer, while the client-side invocation returns after this thread has been created. Given this additional overhead, it is unlikely that *SYNC_NONE* yields a significant performance gain for the client, not even on a multiprocessor machine.

## 16.2 Timing Policies

For each CORBA request four different points in time can be specified:

**Request Start Time** the time after which the request may be delivered to its target

**Request End Time** the time after which the request may no longer be delivered to its target

**Reply Start Time** the time after which the reply may be delivered to the client

**Reply End Time** the time after which the reply may no longer be delivered to the client

Each of these points in time can be specified on a per-object level as a client-side override policy: *RequestStartTimePolicy*, *RequestEndTimePolicy*, *ReplyStartTimePolicy*, and *ReplyEndTimePolicy* (see below for concrete code examples).

Each of these policies specifies an absolute time, which means that they will usually have to be set again for each individual request. As a convenience, there are two additional policies that allow you to specify a *relative* time for *Request End Time* and *Reply End Time*; they are called *RelativeRequestTimeoutPolicy* and *RelativeRoundtripTimeoutPolicy*, respectively. These timeouts are simply more convenient ways for expressing these two times; before each individual invocation, the ORB computes absolute times from them (measured from the start of the invocation at the client side) and handles them just as if an absolute *Request End Time* or *Reply End Time* had been specified. We will therefore only discuss the four absolute timing policies below.

All of these policies apply to synchronous and asynchronous invocations alike.

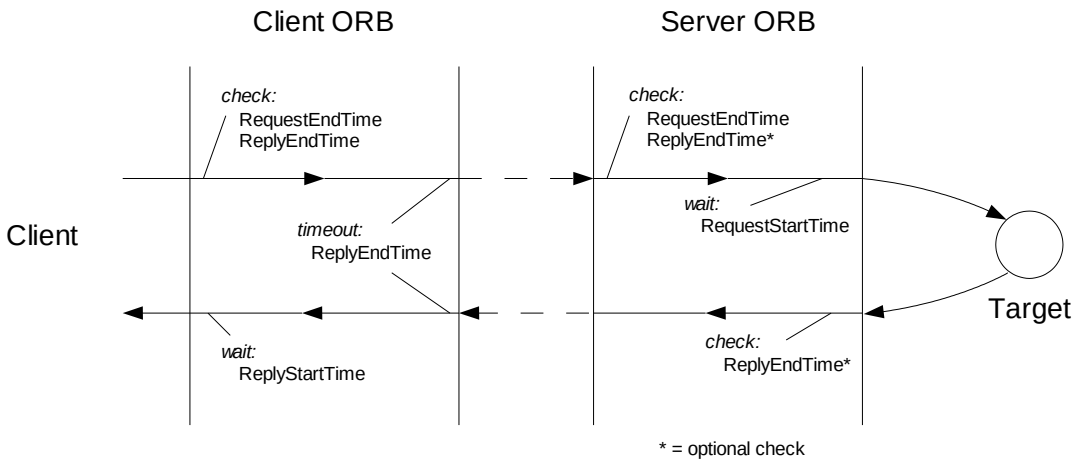


Figure 16.1: Timing Policies in JacORB

Figure 16.1 shows how JacORB interprets the timing policies in the course of a single request.

- As soon as the ORB receives control (prior to marshaling), it converts any *RelativeRequestTimeoutPolicy* or *RelativeRoundtripTimeoutPolicy* to an absolute value, by adding the relative value to the current system time.
- The ORB then checks whether *Request End Time* or *Reply End Time* have already elapsed. If so, no invocation is made, and an `org.omg.CORBA.TIMEOUT` is thrown to the client.
- After the ORB has sent the request, it waits for a reply until *Reply End Time* has elapsed. If it receives no reply before that, the request is discarded and an `org.omg.CORBA.TIMEOUT` thrown to the client. (JacORB does not currently cancel the outstanding request, it simply discards the reply, should one arrive after the timeout has elapsed.)[1]
- On the server side (before demarshaling), the ORB checks whether the *Request End Time* has already elapsed. If so, the request is not delivered to the target, and an `org.omg.CORBA.TIMEOUT` is thrown back to the client.
- Optionally, the server-side ORB may also check at this point whether the *Reply End Time* has already elapsed, and not actually invoke the target in this case (throwing back an `org.omg.CORBA.TIMEOUT` to the client as well). Since the *Reply End Time* would then be checked both on the client and the server side, this requires that the clocks on both machines are synchronized at least to the same order of magnitude as the timeout itself. This check is therefore off by default, and may be enabled by setting the property `jacorb.poa.check_reply_end_time` to "on".
- If the request proceeds, the ORB waits until the *Request Start Time* has been reached, if one was specified, and has not already elapsed. After that, the request is delivered to the target.
- After the target invocation has returned, the ORB may optionally check whether the *Reply End Time* has now elapsed. Similar to the check prior to the target invocation, this check is also optional and controlled by the property `jacorb.poa.check_reply_end_time` (see discussion above). If the check is enabled, and the *Reply End Time* is found to have elapsed at this point, the ORB sends an `org.omg.CORBA.TIMEOUT` back to the client, rather than the actual reply.
- If the reply arrives at the client before *Reply End Time* has elapsed, the ORB waits until *Reply Start Time* has been reached, if one was specified, and has not already elapsed. After that, the reply is delivered back to the client.

The bottom line of this is that for a simple, per-invocation timeout, you should specify a *RelativeRoundtripTimeoutPolicy*.

[1] Note that if there is no connection to the server yet, other timeouts are applied first, configured by the properties `jacorb.connection.client.connect_timeout` and `jacorb.retries`. If connection establishment fails, control does not return to the client until these timeouts have expired, even if this is later than *Reply End Time*.

## Programming

In CORBA, points of time are specified to an accuracy of 100 nanoseconds, using values of struct `TimeBase::UtcT`. To allow easy manipulation of such values from Java, JacORB provides a number of static methods in `org.jacorb.util.Time`. For example, to convert the current Java time into a `UtcT` value, write

```
UtcT currentTime = org.jacorb.util.Time.corbaTime();
```

To create a `UtcT` value that specifies a time $n$ milliseconds in the future, you can write

```
UtcT time = org.jacorb.util.Time.corbaFuture (10000 * n);
```

(The argument to `corbaFuture()` is in CORBA time units of 100 ns; we multiply $n$ by 10000 here to convert it from Java time units (milliseconds).)

The following shows how to set a timing policy for an object using the standard mechanism (see the beginning of this chapter for an explanation). In this example, we set a *Reply End Time* that lies one second in the future:

```
import org.omg.CORBA.*;

SomeCorbaType server  = ...  // the object for which we want to set
                             // a timing policy
org.omg.CORBA.ORB orb = ...
org.omg.CORBA.Any a   = orb.create_any();

org.omg.TimeBase.UtcT replyEndTime
    = org.jacorb.util.Time.corbaFuture (1000 * 10000);  // one second

org.omg.TimeBase.UtcTHelper.insert (a, replyEndTime);

try
{
    Policy p
        = orb.create_policy (REPLY_END_TIME_POLICY_TYPE.value, a);
    server._set_policy_override (new Policy[]{ p },
                                 SetOverrideType.ADD_OVERRIDE);
}
catch (PolicyError e)
{
    ...
}
```

Using the constructors of JacORB's implementations of policy values, this becomes less verbose:

```
SomeCorbaType server  = ...

Policy p = new org.jacorb.orb.policies.ReplyEndTimePolicy
                  (org.jacorb.util.Time.corbaFuture (1000 * 10000));

server._set_policy_override (new Policy[]{ p },
                             SetOverrideType.ADD_OVERRIDE);
```

Likewise, to set a *Relative Roundtrip Timeout* of one second, write:

```
SomeCorbaType server  = ...

Policy p =
    new org.jacorb.orb.policies.RelativeRoundtripTimeoutPolicy
                                                    (1000 * 10000);

server._set_policy_override (new Policy[]{ p },
                             SetOverrideType.ADD_OVERRIDE);
```

The difference between this and the example before, where a *Reply End Time* was used, is that the latter specifies a *relative time* to CORBA. The policy will therefore be valid for all subsequent invocations, because the absolute deadline will be recomputed before each invocation. In the first example, the deadline will no longer make sense for any subsequent invocations, since only an absolute time was specified to the ORB.

# 17 Connection Management and Connection Timeouts

JacORB offers a certain level of control over connections and timeouts. You can

- set connection idle timeouts.
- set request timing.
- set the maximum number of accepted TCP/IP connections on the server.

## 17.1 Timeouts

Connection idle timeouts can be set individually for the client and the server. They control how long an idle connection, i.e. a connection that has no pending replies, will stay open. The corresponding properties are `jacorb.connection.client.idle_timeout` and `jacorb.connection.server.timeout` and take their values as milliseconds. If not set, connections will stay open indefinitely (or until the OS decides to close them).

*Request timing* controls how long an individual request may take to complete. The programmer can specify this using QoS policies, discussed in chapter 16.

## 17.2 Connection Management

When a client wants to invoke a remote object, it needs to send the request over a connection to the server. If the connection isn't present, it has to be created. In JacORB, this will only happen once for every combination of host name and port. Once the connection is established, all requests and replies between client and server will use the same connection. This saves resources while adding a thin layer of necessary synchronization, and is the recommended approach of the OMG. Occasionally people have requested to allow for multiple connections to the same server, but nobody has yet presented a good argument that more connections would speed up things considerably.

On the server side, the property `jacorb.connection.max_server_connection` allows to set the maximum number of TCP/IP connections that will be listened on for requests. When using a network sniffer or tools like netstat, more inbound TCP/IP connections than the configured number may be displayed. This is for the following reason: Whenever the connection limit is reached, JacORB tries to close

existing idle connections (see the subsection below). This is done on the thread that accepts the new connections, so JacORB will not actively accept more connections. However, the ServerSocket is initialized with a backlog of 20. This means that 20 more connections will be quasi-accepted by the OS. Only the 21st will be rejected right away.

### 17.2.1 Basics and Design

Whenever there is the need to close an existing connection because of the connection limit, the question arises on which of the connection to close. To allow for maximum flexibility, JacORB provides the interface `SelectionStrategy` that allows for a custom way to select a connection to close. Because selecting a connection usually requires some sort of statistical data about it, the interface `StatisticsProvider` allows to implement a class that collects statistical data.

```
package org.jacorb.orb.giop;

public interface SelectionStrategy
{
    public ServerGIOPConnection
        selectForClose( java.util.List connections );
}

public interface StatisticsProvider
{
    public void messageChunkSent( int size );
    public void flushed();
    public void messageReceived( int size );
}
```

The interface `SelectionStrategy` has only the single method of `selectForClose()`. This is called by the class `GIOPConnectionManager` when a connection needs to be closed. The argument is a `List` containing objects of type `ServerGIOPConnection`. The call itself is synchronized in the `GIOPConnectionManager`, so no additional synchronization has to be done by the implementor of `SelectionStrategy`. When examining the connections, the strategy can get hold of the `StatisticsProvider` via the method `getStatisticsProvider()` of the class `GIOPConnection`. The strategy implementor should take care only to return idle connections. While the connection state is checked anyway while closing (it may have changed in the meantime), it seems to be more efficient to avoid cycling through the connections. When no suitable connection is available, the strategy may return `null`. The `GIOPConnectionManager` will then wait for a configurable time, and try again. This goes on until a connection can be closed.

The interface `StatisticsProvider` is used to collect statistical data about a connection and provide it to the `SelectionStrategy`. Because the nature of this data may vary, there is no standard access to the data via the interface. Therefore, `StatisticsProvider` and `SelectionStrategy` usually need to be implemented together. Whenever a new connection is cre-

ated[1], a new `StatisticsProvider` object is instanciated and stored with the `GIOPConnection`[2]. The `StatisticsProvider` interface is oriented along the mode of use of the `GIOPConnection`. For efficiency reasons, messages are not sent as one big byte array. Instead, they are sent piecewise over the wire. When such a chunk is sent, the method `messageChunkSent(int size)` will be called. After the message has been completely sent, method `flush()` is called. This whole process is synchronized, so all consecutive `messageChunkSent`s until a `flush()` form a single message. Therefore, no synchronization on this level is necessary. However, access to gathered statistical data by the `SelectionStrategy` is concurrent, so care has to be taken. Receiving messages is done only on the whole, so there exists only one method, `messageReceived(int size)`, to notify the `StatisticsProvider` of such an event.

JacORB comes with two pre-implemented strategies: least frequently used and least recently used. LFU and LRU are implemented by the classes `org.jacorb.orb.giop.L[F|R]USelection-StrategyImpl` and `org.jacorb.orb.giop. L[F|R]UStatisticsProviderImpl`.

### 17.2.2 Configuration

To configure connection management, the following properties are provided:

`jacorb.connection.max_server_connections` This property sets the maximum number of TCP/IP connections that will be listened on by the server–side ORB.

`jacorb.connection.wait_for_idle_interval` This property sets the interval to wait until the next try is made to find an idle connection to close. Value is in microseconds.

`jacorb.connection.selection_strategy_class` This property sets the `Selection-Strategy`.

`jacorb.connection.statistics_provider_class` This property sets the `Statistics-Provider`.

`jacorb.connection.delay_close` If turned on, JacORB will delay closing of TCP/IP connections to avoid certain situations, where message loss can occur. See also section 17.2.3.

### 17.2.3 Limitations

When trying to close a connection, it is first checked that the connection is idle, i.e. has no pending messages. If this is the case, a GIOP CloseConnection message is sent, and the TCP/IP connection is closed. Under high load, this can lead to the following situation:

1. Server sends the CloseConnection message.
2. Server closes the TCP/IP connection.

[1] Currently, connection management is only implemented for the server side. Therefore, only accepted `ServerGIOPConnection`s will get a `StatisticsProvider`

[2] This is actually only done when a `StatisticsProvider` is configured

3. The client sends a new request into the connection, because it hasn't yet read and acted on the CloseConnection message.
4. The server–side OS will send a TCP RST, which cancels out the CloseConnection message.
5. The client finds the connection closed and must consider the request lost.

To get by this situation, JacORB takes the following approach. Instead of closing the connection right after sending the CloseConnection message, we delay closing and wait for the client to close the connection. This behaviour is turned off by default, but can be enabled by setting the property `jacorb.connection.delay_close` to "yes". When non-JacORB clients are used care has to be taken that these ORBs do actively close the connection upon receiving a CloseConnection message.

# 18 Extensible Transport Framework

The *Extensible Transport Framework (ETF)*, which JacORB implements, allows you to plug in other transport layers besides the standard IIOP (TCP/IP) protocol[1].

To use an alternative transport, you need to (a) implement it as a set of Java classes following the ETF specification, and (b) tell JacORB to use the new transport instead of (or alongside with) the standard IIOP transport. We cover both steps below.

## 18.1 Implementing a new Transport

The interfaces that an ETF-compliant transport must implement are described in the ETF specification, and there is thus no need to repeat that information here. JacORB's default IIOP transport, which is realized in the package `org.jacorb.orb.iiop`, can also serve as a starting point for implementing your own transports.

For each transport, the following interfaces must be implemented (defined in `ETF.idl`, the package is `org.omg.ETF`):

**Profile** encapsulates addressing information for this transport

**Listener** server-side communication endpoint, waits for incoming connections and passes them up to the ORB

**Connection** an actual communication channel for this transport

**Factories** contains factory methods for the above interfaces

The `Handle` interface from the ETF package is implemented in the ORB (by the class `org.jacorb.orb.BasicAdapter`), not by individual transports. There is currently no support in JacORB for the optional zero-copy mechanism; the interface `ConnectionZeroCopy` therefore needn't be implemented.

On the server side, the `Listener` must pass incoming connections up to the ORB using the "Handle" mechanism; the `accept()` method needn't be implemented. Once a `Connection` has been passed up to the ORB, it will never be "returned" to the `Listener` again. The method `completed_data()` in the `Listener` interface therefore needn't be implemented, and neither should the `Listener` ever call `Handle.signal_data_available()` or `Handle.closed_by_peer()` (these methods throw a `NO_IMPLEMENT` exception in JacORB).

[1] At the time of this writing (July 2003), ETF is still a draft standard (OMG TC document mars/2003-02-01).

At the time of this writing (July 2003), there is still uncertainty in ETF about how server-specific Profiles (as returned by `Listener.endpoint()`, for example) should be turned into object-specific ones for inclusion into IORs. We have currently added three new operations to the `Profile` interface to resolve this issue, see JacORB's version of `ETF.idl` for details.

## 18.2 Configuring Transport Usage

You tell JacORB which transports it should use by listing the names of their `Factories` classes in the property `jacorb.transport.factories`. In the standard configuration, this property contains only `org.jacorb.orb.iiop.IIOPFactories`, the `Factories` class for the standard IIOP transport. The property's value is a comma-separated list of fully qualified Java class names; each of these classes must be found somewhere on the CLASSPATH that JacORB is started with. For example:

```
jacorb.transport.factories = my.transport.Factories, org.jacorb.orb.iiop.IIOPFactories
```

By default, a JacORB server creates listeners for each transport listed in the above property, and publishes profiles for each of these transports in any IOR it creates. The order of profiles within an IOR is the same as that of the transports in the property.

If you don't want your servers to listen on each of these transports (e.g. because you want some of your transports only to be used for client-side connections), you can specify the set of actual listeners in the property `jacorb.transport.server.listeners`. The value of this property is a comma-separated list of numeric profile tags, one for each transport that you want listeners for, and which you want published in IOR profiles. The numeric value of a transport's profile tag is the value returned by the implementation of `Factories.profile_tag()` for that transport. Standard IIOP has profile tag 0 (`TAG_INTERNET_IOP`). Naturally, you can only specify profile tag numbers here for which you have a corresponding entry in `jacorb.transport.factories`.

So, to restrict your server-side transports to standard IIOP, you would write:

```
jacorb.transport.server.listeners = 0
```

On the client side, the ORB must decide which of potentially many transports it should use to contact a given server. The default strategy is that for each IOR, the client selects *the first profile for which there is a transport implementation available at the client side* (specified in `jacorb.transport.factories`). Profiles for which the client has no transport implementation are skipped.

Note that this is a purely static decision, based on availability of an implementation. JacORB does not attempt to actually establish a transport connection in order to find out which transport can be used. Also, should the selected transport fail, JacORB does not "fall back" to the next transport in the list. (This is because JacORB opens connections lazily, only when the first actual data is being sent.)

You can customize this strategy by providing your own implementation of `org.jacorb.orb.ProfileSelector`, and specifying it in the property `jacorb.transport.client.selector`. The interface `ProfileSelector` requires a single method,

```
public Profile selectProfile (List profiles,
                              ClientConnectionManager ccm);
```

For each IOR, this method receives a list of all profiles from the IOR for which the client has a transport implementation, in the order in which they appear in the IOR. The method should select one profile from this list and return it; this profile will then be used for communication with the server.

To help with the decision, JacORB's `ClientConnectionManager` is passed as an additional parameter. The method implementation can use it to check whether connections with a given transport, or to a given server, have already been made; it can also try and pre-establish a connection using a given transport and store it in the `ClientConnectionManager` for later use. (See the JacORB source code to find out how to deal with the `ClientConnectionManager`.)

The default `ProfileSelector` does not use the `ClientConnectionManager`, it simply returns the first profile from the list, unconditionally. To let JacORB use your own implementation of the `ProfileSelector` interface, specify the fully qualified classname in the property:

```
jacorb.transport.client.selector=my.pkg.MyProfileSelector
```

## 18.3 Selecting Specific Profiles Using RT Policies

JacORB has a implementation of the standard Real Time CORBA ClientProtocolPolicy policy which it uses to allow a developer to select between IIOP profiles that either support or do not support an SSL component. When applied to a bind (implicit or explicit), the ClientProtocolPolicy indicates the protocols that may be used to make a connection to the specified object.

The only non-standard proprietary component of this is the definition of two profile IDs that are used to distinguish between IIOP/SSL, IIOP/NOSSL and IIOP profiles. The three `org.omg.RTCORBA.Protocol` types are:

- `JAC_SSL_PROFILE_ID`
- `NOSSL_PROFILE_ID`
- `org.omg.IOP.TAG_INTERNET_IOP`

The former two are defined within `org.jacorb.orb.ORBConstants`. To apply this the developer may use, for example, a ClientRequestInterceptor that applies the policy to the object and throws a

ForwardRequest, or may simply apply the policy to the object as shown below.

```
org.omg.RTCORBA.Protocol protocol = new org.omg.RTCORBA.Protocol();
org.omg.RTCORBA.Protocol protocols[] = new org.omg.RTCORBA.Protocol[1];
org.omg.CORBA.Policy policies[] = new org.omg.CORBA.Policy[1];

protocol.protocol_type = ORBConstants.JAC_SSL_PROFILE_ID;
protocols[0] = protocol;

rtorb = org.omg.RTCORBA.RTORBHelper.narrow
            (orb.resolve_initial_references ("RTORB"));

org.omg.RTCORBA.ClientProtocolPolicy cpp =
            rtorb.create_client_protocol_policy (protocols);

policies[0] = cpp;

<mycorbaobject>._set_policy_override
            (policies, SetOverrideType.SET_OVERRIDE);
```

# 19 Security Attribute Service

The Security Attribute Service (SAS) is part of the Common Secure Interoperability Specification, Version 2 (CSIv2) CORBA specification. It is defined in the Secure Interoperability chapter (chapter 24) of the CORBA 3.0.2 Specification.

## 19.1 Overview

The SAS specification defines the interchange between a Client Security Service (CSS) and a Target Security Service (TSS) for the exchange of security authentication and authorization elements. This information is exchanged in the Service Context of the GIOP request and reply messages. The SAS may be used in conjunction with SSL to provide privacy of the messages being sent and received.

The SAS service is implemented as a series of standard CORBA interceptors, one for the CSS and one for the TSS. The service also uses a user specified SAS context class to support different authentication mechanisms, such as GSSUP and Kerberos.

The SAS service is activated based on entries in the JacORB properties file and CORBA Properties assigned to the POA.

The following is a part of the JacORB properties file that is used by the SAS.

```
########################################
#                                      #
#   SAS configuration                  #
#                                      #
########################################

jacorb.SAS.log.verbosity=INFO
jacorb.SAS.CSS.log.verbosity=INFO
jacorb.SAS.TSS.log.verbosity=INFO

# This option defines the specific SAS context generator/validator
# Currently supported contexts include:
#    GssUpContext      - Uses GSSUP security
#    KerberosContext   - uses Kerberos security
# At least one context must be selected for SAS support
jacorb.security.sas.contextClass=org.jacorb.security.sas.GssUpContext
#jacorb.security.sas.contextClass=org.jacorb.security.sas.KerberosContext

# This initializer installs the SAS interceptors
# Comment out this line if you do not want SAS support
org.omg.PortableInterceptor.ORBInitializerClass.SAS=org.jacorb.security.sas.SASInitializer

# This option is used for GSSUP security and sets up the GSS Provider
# Comment out this line if you are not using GSS UP authentication
```

```
org.omg.PortableInterceptor.ORBInitializerClass.GSSUPProvider=org.jacorb.security.sas.GSSUPProviderInitializer
```

## 19.2 GSSUP Example

The GSSUP (GSS Username/Password) example demonstrates the simplest usage of the SAS service. In this example, username and password pairs are send via the SAS service. The client registers its username and password with the GSSUP Context which is later used CSS interceptor to generate the user's authentication information. The TSS retrieves the username and password without validating them. It is assumed by the TSS that the username and password are correct and/or will be further validated by a later interceptor or application code.

The following describes a SAS example using GSSUP.

### 19.2.1 GSSUP IDL Example

```
module demo{
  module sas{
    interface SASDemo{
      void printSAS();
    };
  };
};
```

The IDL contains a single interface. This interface is used to print out the user principal sent and received by the SAS service.

### 19.2.2 GSSUP Client Example

The following is a sample GSSUP client.

```
package demo.sas;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;

import org.jacorb.security.sas.GssUpContext;
import org.omg.CORBA.ORB;

public class GssUpClient {
        public static void main(String args[]) {
                if (args.length != 3) {
                        System.out.println("Usage: java demo.sas.GssUpClient <ior_file> <username> <password>"
                        System.exit(1);
                }

                try {
                        // set security credentials
                        GssUpContext.setUsernamePassword(args[1], args[2]);
```

```
                        // initialize the ORB.
                        ORB orb = ORB.init(args, null);

                        // get the server
                        File f = new File(args[0]);
                        if (!f.exists()) {
                                System.out.println("File " + args[0] + " does not exist.");
                                System.exit(-1);
                        }
                        if (f.isDirectory()) {
                                System.out.println("File " + args[0] + " is a directory.");
                                System.exit(-1);
                        }
                        BufferedReader br = new BufferedReader(new FileReader(f));
                        org.omg.CORBA.Object obj = orb.string_to_object(br.readLine());
                        br.close();
                        SASDemo demo = SASDemoHelper.narrow(obj);

                        //call single operation
                        demo.printSAS();
                        demo.printSAS();
                        demo.printSAS();

                        System.out.println("Call to server succeeded");
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
        }
}
```

The key to the client is the call to:

```
GssUpContext.setUsernamePassword(args[1], args[2]);
```

This call registers the client's username and password with the GSSUP context. This information will then later be used by the CSS interceptor as the user's authentication information.

### 19.2.3 GSSUP Target Example

The following is a sample GSSUP target.

```
package demo.sas;

import java.io.FileWriter;
import java.io.PrintWriter;

import org.jacorb.sasPolicy.SASPolicyValues;
import org.jacorb.sasPolicy.SAS_POLICY_TYPE;
import org.jacorb.sasPolicy.SASPolicyValuesHelper;
import org.omg.PortableServer.IdAssignmentPolicyValue;
import org.omg.PortableServer.LifespanPolicyValue;
import org.omg.PortableServer.POA;
import org.omg.CORBA.ORB;
import org.omg.CORBA.Any;
import org.omg.CSIIOP.EstablishTrustInClient;

public class GssUpServer extends SASDemoPOA {
```

```
        private ORB orb;

        public GssUpServer(ORB orb) {
                this.orb = orb;
        }

        public void printSAS() {
                try {
                        org.omg.PortableInterceptor.Current current = (org.omg.PortableInterceptor.Current)orb
                        org.omg.CORBA.Any anyName = current.get_slot(org.jacorb.security.sas.SASInitializer.sa
                        if( anyName.type().kind().value() == org.omg.CORBA.TCKind._tk_null ) {
                                System.out.println("Null Name");
                        } else {
                                String name = anyName.extract_string();
                                System.out.println("printSAS for user " + name);
                        }
                } catch (Exception e) {
                        System.out.println("printSAS Error: " + e);
                }
        }

        public static void main(String[] args) {
                if (args.length != 1) {
                        System.out.println("Usage: java demo.sas.GssUpServer <ior_file>");
                        System.exit(-1);
                }

                try {
                        // initialize the ORB and POA.
                        ORB orb = ORB.init(args, null);
                        POA rootPOA = (POA) orb.resolve_initial_references("RootPOA");
                        org.omg.CORBA.Policy [] policies = new org.omg.CORBA.Policy[3];
                        policies[0] = rootPOA.create_id_assignment_policy(IdAssignmentPolicyValue.USER_ID);
                        policies[1] = rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT);
                        Any sasAny = orb.create_any();
                        SASPolicyValuesHelper.insert( sasAny, new SASPolicyValues(EstablishTrustInClient.value
                        policies[2] = orb.create_policy(SAS_POLICY_TYPE.value, sasAny);
                        POA securePOA = rootPOA.create_POA("SecurePOA", rootPOA.the_POAManager(), policies);
                        rootPOA.the_POAManager().activate();

                        // create object and write out IOR
                        GssUpServer server = new GssUpServer(orb);
                        securePOA.activate_object_with_id("SecureObject".getBytes(), server);
                        org.omg.CORBA.Object demo = securePOA.servant_to_reference(server);
                        PrintWriter pw = new PrintWriter(new FileWriter(args[0]));
                        pw.println(orb.object_to_string(demo));
                        pw.flush();
                        pw.close();

                        // run the ORB
                        orb.run();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

# 19.3 Kerberos Example

The Kerberos example demonstrates how to integrate the use of a kerberos service to provide authentication credentials to the SAS service. In this example, the Java(TM) Authentication and Authorization Service (JAAS) is used to perform the Kerberos login and to return the principal and Kerberos ticket. The actual username and password may either be entered by the user or derived from the current user's Kerberos login session. For Windows 2000 Active Directory networks, this means that the user's credentials can be automatically obtained from the Windows login.

The following describes a SAS example using Kerberos.

## 19.3.1 Kerberos IDL Example

```
module demo{
  module sas{
    interface SASDemo{
      void printSAS();
    };
  };
};
```

The IDL contains a single interface. This interface is used to print out the user principal sent and received by the SAS service.

## 19.3.2 Kerberos Client Example

The following is a sample Kerberos client.

```
package demo.sas;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.security.Principal;
import java.security.PrivilegedAction;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

import org.omg.CORBA.ORB;

public class KerberosClient {
        private static Principal myPrincipal = null;
        private static Subject mySubject = null;
        private static ORB orb = null;

        public KerberosClient(String args[]) {

                try {
                        // initialize the ORB.
                        orb = ORB.init(args, null);
```

```
                // get the server
                File f = new File(args[0]);
                if (!f.exists()) {
                        System.out.println("File " + args[0] + " does not exist.");
                        System.exit(-1);
                }
                if (f.isDirectory()) {
                        System.out.println("File " + args[0] + " is a directory.");
                        System.exit(-1);
                }
                BufferedReader br = new BufferedReader(new FileReader(f));
                org.omg.CORBA.Object obj = orb.string_to_object(br.readLine());
                br.close();
                SASDemo demo = SASDemoHelper.narrow(obj);

                //call single operation
                demo.printSAS();
                demo.printSAS();
                demo.printSAS();

                System.out.println("Call to server succeeded");
        } catch (Exception ex) {
                ex.printStackTrace();
        }
}

public static void main(String args[]) {
        if (args.length != 3) {
                System.out.println("Usage: java demo.sas.KerberosClient <ior_file> <username> <passwor
                System.exit(1);
        }

        // login - with Kerberos
        LoginContext loginContext = null;
        try {
                JaasTxtCalbackHandler txtHandler = new JaasTxtCalbackHandler();
                txtHandler.setMyUsername(args[1]);
                txtHandler.setMyPassword(args[2].toCharArray());
                loginContext = new LoginContext("KerberosClient", txtHandler);
                loginContext.login();
        } catch (LoginException le) {
                System.out.println("Login error: " + le);
                System.exit(1);
        }
        mySubject = loginContext.getSubject();
        myPrincipal = (Principal) mySubject.getPrincipals().iterator().next();
        System.out.println("Found principal " + myPrincipal.getName());

        // run in privileged mode
        final String[] finalArgs = args;
        try {
                Subject.doAs(mySubject, new PrivilegedAction() {
                        public Object run() {
                                try {
                                        KerberosClient client = new KerberosClient(finalArgs);
                                        orb.run();
                                } catch (Exception e) {
                                        System.out.println("Error running program: "+e);
                                }
                                System.out.println("Exiting privileged operation");
                                return null;
                        }
                });
        } catch (Exception e) {
                System.out.println("Error running privileged: "+e);
```

```
                }
        }
}
```

The CSS uses JAAS to logon and return the user's Kerberos credentials. The CSS must then run the rest of the application as a PrivilegedAction using the logged on credentials. This allows the CSS interceptor to retrieve the Kerberos ticket from the logon session.

The following is the JAAS logon configuration for the CSS:

```
KerberosClient
{
    com.sun.security.auth.module.Krb5LoginModule required storeKey=true useTicketCache=true debug=true;
};
```

### 19.3.3 Kerberos Target Example

The following is a sample Kerberos target.

```
package demo.sas;

import java.io.FileWriter;
import java.io.PrintWriter;
import java.security.Principal;
import java.security.PrivilegedAction;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

import org.jacorb.sasPolicy.SASPolicyValues;
import org.jacorb.sasPolicy.SAS_POLICY_TYPE;
import org.jacorb.sasPolicy.SASPolicyValuesHelper;
import org.omg.PortableServer.IdAssignmentPolicyValue;
import org.omg.PortableServer.LifespanPolicyValue;
import org.omg.PortableServer.POA;
import org.omg.CORBA.ORB;
import org.omg.CORBA.Any;
import org.omg.CSIIOP.EstablishTrustInClient;

public class KerberosServer extends SASDemoPOA {
        private static Principal myPrincipal = null;
        private static Subject mySubject = null;
        private ORB orb;

        public KerberosServer(ORB orb) {
                this.orb = orb;
        }

        public void printSAS() {
                try {
                        org.omg.PortableInterceptor.Current current = (org.omg.PortableInterceptor.Current) or
                        org.omg.CORBA.Any anyName = current.get_slot(org.jacorb.security.sas.SASInitializer.sa
                        String name = anyName.extract_string();
                        System.out.println("printSAS for user " + name);
                } catch (Exception e) {
                        System.out.println("printSAS Error: " + e);
                }
```

```
}

public KerberosServer(String[] args) {
        try {
                // initialize the ORB and POA.
                orb = ORB.init(args, null);
                POA rootPOA = (POA) orb.resolve_initial_references("RootPOA");
                org.omg.CORBA.Policy [] policies = new org.omg.CORBA.Policy[3];
                policies[0] = rootPOA.create_id_assignment_policy(IdAssignmentPolicyValue.USER_ID);
                policies[1] = rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT);
                Any sasAny = orb.create_any();
                SASPolicyValuesHelper.insert( sasAny, new SASPolicyValues(EstablishTrustInClient.value
                policies[2] = orb.create_policy(SAS_POLICY_TYPE.value, sasAny);
                POA securePOA = rootPOA.create_POA("SecurePOA", rootPOA.the_POAManager(), policies);
                rootPOA.the_POAManager().activate();

                // create object and write out IOR
                securePOA.activate_object_with_id("SecureObject".getBytes(), this);
                org.omg.CORBA.Object demo = securePOA.servant_to_reference(this);
                PrintWriter pw = new PrintWriter(new FileWriter(args[0]));
                pw.println(orb.object_to_string(demo));
                pw.flush();
                pw.close();
        } catch (Exception e) {
                e.printStackTrace();
        }
}

public static void main(String[] args) {
        if (args.length != 2) {
                System.out.println("Usage: java demo.sas.KerberosServer <ior_file> <password>");
                System.exit(-1);
        }

        // login - with Kerberos
        LoginContext loginContext = null;
        try {
                JaasTxtCalbackHandler cbHandler = new JaasTxtCalbackHandler();
                cbHandler.setMyPassword(args[1].toCharArray());
                loginContext = new LoginContext("KerberosService", cbHandler);
                loginContext.login();
        } catch (LoginException le) {
                System.out.println("Login error: " + le);
                System.exit(1);
        }
        mySubject = loginContext.getSubject();
        myPrincipal = (Principal) mySubject.getPrincipals().iterator().next();
        System.out.println("Found principal " + myPrincipal.getName());

        // run in privileged mode
        final String[] finalArgs = args;
        try {
                Subject.doAs(mySubject, new PrivilegedAction() {
                        public Object run() {
                                try {
                                        // create application
                                        KerberosServer app = new KerberosServer(finalArgs);
                                        app.orb.run();
                                } catch (Exception e) {
                                        System.out.println("Error running program: "+e);
                                }
                                return null;
                        }
                });
        } catch (Exception e) {
```

```
                        System.out.println("Error running privileged: "+e);
                }
        }
}
```

The TSS uses JAAS to logon and return the user's Kerberos credentials. The logon principal to use is defined in the JAAS login configuration file. The TSS must then run the rest of the application as a PrivilegedAction using the logged on credentials. This allows the TSS interceptor to retrieve the Kerberos ticket from the logon session.

The following is the JAAS logon configuration for the TSS:

```
KerberosService
{
    com.sun.security.auth.module.Krb5LoginModule required storeKey=true principal="testService@OPENROADSCONSUL
};
```

# 20 The JacORB Notification Service

The JacORB Notification Service is a partial implementation of the Notification Service specified by the OMG.

## 20.1 Unsupported Features

The JacORB Notification Service does not support persistent connections or events.

## 20.2 Installation

### 20.2.1 JDK 1.3

If you're using JDK 1.3 and want to use the JacORB Notification Service you'll need to download the additional library gnu.regexp from http://www.cacas.org/java/gnu/regexp and put it in your classpath. This is necessary because the JacORB Notification Service uses regular expressions. Regular expressions are available in the JDK since version 1.4. Alternatively you can download Jakarta Regexp http://jakarta.apache.org/regexp.

## 20.3 Running the Notification Service

Before the JacORB Notification Service can be accessed, a server process must be started. Starting the notification server is done by running

```
$ ntfy [-printIOR] [-printCorbaloc] [-writeIOR filename]
[-registerName nameID[.nameKind]] [-port oaPort] [-channels channels]
[-help]
```

| | |
|---|---|
| -printIOR | print the IOR to STDOUT |
| -printCorbaloc | print the Corbaloc to STDOUT |
| -writeIOR *filename* | write the IOR to a file |
| -registerName *nameId*[.*nameKind*] | make a Name Service entry for the EventChannelFactory. The Notification Service will resolve the Name Service by invoking `resolve_initial_references("NameService")`. |

| | |
|---|---|
| | Ensure that your environment is set up properly. |
| -port *oaport* | start the Notification Service on the specified port. |
| -channels *channels* | create a number of EventChannels. |

## 20.3.1 Running as a NT Service or an UNIX Daemon

With a little help from the Java Service Wrapper it is easy to run the JacORB notification service as a NT Service or as an UNIX daemon.

### Note for JDK 1.3 Users

As noted if you are running JDK 1.3 you need to provide an additional library. If you use the wrapper you also need to add a classpath entry to the wrapper configuration file.

Edit `bin/NotifyService-Wrapper.conf` and add a classpath entry:

```
# Java Classpath (include wrapper.jar)  Add class path elements as
#  needed starting from 1
wrapper.java.classpath.1=../lib/wrapper-3.x.y.jar
...
wrapper.java.classpath.6=../lib/avalon-framework-4.1.5.jar
wrapper.java.classpath.7=../lib/gnu.regexp.jar
```

### Installing and Running as a NT Service

The necessary wrapper configuration files are located in the `JacORB/bin` directory.

The notification service can be installed as a NT service by double clicking on the `NotifyService-Install-NT.bat` batch file which is located in the `JacORB/bin` directory. Alternatively you can open a Command Window and then run the install script from the command prompt.

```
C:\JacORB\bin>NotifyService-Install-NT.bat
wrapper  | JacORB Notification Service installed.
```

Once the service has been installed, it can be started by opening up the Service Control Panel, selecting the service, and then pressing the start button.

The service can also be started and stopped from within a Command Window by using the `net start JacORB-Notify` and `net stop JacORB-Notify` commands, or by passing commands to the wrapper.exe executable.

The wrapper is set up to start the JacORB Notification Service whenever the machine is rebooted. The service can be uninstalled by running the `NotifyService-Uninstall-NT.bat` batch file. See the Windows specific wrapper documentation for more details.

### Installing and Running as an UNIX Daemon

JacORB is shipped with a `sh` script which can be used to start and stop the JacORB Notification Service controlled by the Java Service Wrapper.

First you need to download the appropiate binary for your system from http://wrapper.tanukisoftware.org. The Java Service Wrapper is supported on Windows, Linux, Solaris, AIX, HP-UX, Macintosh OS X, DEC OSF1, FreeBSD, and SGI Irix systems (Note: You don't need to download anything if you are running Windows. All necessary stuff is shipped with the JacORB distribution).

Install the Java Service Wrapper to a appropiate place by unzipping it (*WRAPPER_HOME*). Add *WRAPPER_HOME*`/bin` to your PATH variable. If you don't want to modify your PATH variable you can put a link to *WRAPPER_HOME*`/bin/wrapper` in one of the directories that's already in your PATH environment (e.g. `ln -s /usr/local/wrapper/bin/wrapper /usr/local/bin`).

Ensure that the shell-script `JacORB/bin/ntfy-wrapper` has the executable bit set. Note that the .sh script will attempt to create a pid file in the directory specified by the property `PIDDIR` in the script. If the user used to launch the Wrapper does not have permission to write to this directory then this will result in an error. An alternative that will work in most cases is to write the pid file to another directory. To make this change, edit the .sh script and change the following setting:

```
PIDDIR="."
```

to something more appropiate:

```
PIDDIR="/var/run"
```

**Running in the console** The JacORB notification service can now be run by simply executing `bin/ntfy-wrapper console`. When running using the console command, output from the notification service will be visible in the console. The notification service can be terminated by hitting CTRL-C in the command window. This will cause the Wrapper to shut down the service cleanly.

If you omit the command the scripts prints the available commands. The script accepts the commands start, stop, restart and dump. The start, stop, and restart commands are common to most daemon scripts and are used to control the wrapper and the notification service as a daemon process. The console command will launch the wrapper in the current shell, making it possible to kill the application with CTRL-C. Finally the command dump will send a kill -3 signal to the wrapper causing its JVM to do a full thread dump.

**Running as a Daemon Process** The application can be run as a detatched daemon process by executing the script using the *start* command.

When running using the start command, output from the JVM will only be visible by viewing the logfile `NotifyService-Wrapper.log` using `tail -f NotifyService-Wrapper.log`. The location of the logfile can be configured in the wrapper configuration file `bin/NotifyService-Wrapper.conf`

Because the application is running as a detatched process, it can not be terminated using CTRL-C and will continue to run even if the console is closed.

To stop the application rerun the script using the *stop* command.

**Installing The Notification Service To Start on Reboot** This is system specific. See the UNIX specific wrapper documentation for instructions for some platforms.

### 20.3.2 Running as a JBoss Service

The JacORB notification service can also be run as a jboss service.

As first step the jacorb.jar that is shipped with jboss (`JBOSS_HOME/server/all/lib`) needs to be replaced to the current version (v2.2.3). As next step the jboss-cosnotification.sar file can be deployed into jboss by copying it to `JBOSS_HOME/server/all/deploy`.

After starting jboss a mbean for the notification service will show up in the jboss jmx management console.

## 20.4 Accessing the Notification Service

Configuring a default notification service as the ORB's default is done by adding the URL that points to the service to the properties files `.jacorb_properties`. A valid URL can be obtained in various ways:

1. By specifying the option `-printIOR` as you start the notification service a stringified IOR is printed out to the console. From there you can copy it to a useful location.
2. Usually the stringified IOR makes most sense inside a file. Use the option `-writeIOR <filename>` to write the IOR to the specified file.
3. A more compact URL can be obtained by using the option `-printCorbaloc`. In conjunction with the option `-port` you can use the simplified corbaloc: URL of the form `corbaloc::ip-address:port/NotificationService`. This means all you need to know to construct an object reference to your notification service is the IP address of the machine and the port number the server process ist listening on (the one specified using `-port`).

Add the property `ORBInitRef.NotificationService` to your properties file. The value can be a corbaloc: URL or alternatively the file name where you saved the IOR.

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

```
// get a reference to the notification service
ORB orb = ORB.init(args, null);
org.omg.CORBA.Object obj;
```

```
obj = orb.resolve_initial_references("NotificationService");
EventChannelFactory ecf = EventChannelFactoryHelper.narrow( o );
IntHolder ih = new IntHolder();
Property[] p1 = new Property[0];
Property[] p2 = new Property[0];
EventChannel ec = ecf.create_channel(p1, p2, ih);
...
```

## 20.5 Configuration

Following is a brief description of the properties that control Notification Service behaviour.

The Notification Service uses up to three Thread Pools with a configurable size. The first Thread Pool is used to process the filtering of the Messages. The second Thread Pool is used to deliver the Messages to the Consumers. The third Thread Pool us used to pull Messages from PullSuppliers.

Table 20.1: Notification Service Properties

| Property | Description | Type | Default |
|---|---|---|---|
| `filter.thread_pool_size`[1] | This is the Size of the Thread Pool used to process the filters. Increasing this value on a Multiprocessor machine or if Filters are on a different machine than the Channel could increase the Filtering Performance as multiple events can be processed concurrently. | int $\geq 0$ | 2 |
| `proxysupplier.thread_pool_size` | This is the Size of the Thread Pool used to deliver the Messages to the Consumers. By using the property `proxysupplier.threadpolicy`[2] it is also possible to use one Thread per ProxySupplier. | int $\geq 0$ | 4 |
| `proxyconsumer.thread_pool_size` | Specifies the Size of the Thread Pool used to pull Messages from PullSuppliers | int $>= 0$ | 2 |

[1] All notification service properties share the common prefix *jacorb.notification* which is omitted here to save some space

[2] also abbreviated.

Table 20.1: Notification Service Properties

| Property | Description | Type | Default |
|---|---|---|---|
| `proxysupplier. threadpolicy` | Specify which thread policy the ProxySuppliers should use to deliver the Messages to its Consumers. Valid values are:<br>**ThreadPool** a fixed number of threads is used. See property `proxysupplier. thread_pool_size`.<br>**ThreadPerProxy** Each ProxySupplier uses its own thread. | string | Thread-Pool |
| `supplier. poll_intervall` | Specifies how often Messages should be pulled from a PullSupplier. The value specifies the intervall between two pull-Operations. | milli-seconds | 1000 |
| `supplier. max_number` | Specify the maximum number of Suppliers that may be connected to a Channel at a time. If a Supplier tries to connect, while this limit is exceeded, AdminLimitExceeded is raised. Note that this property can also be set programatically via the `set_admin` operation. | int $> 0$ | maximum int value |
| `consumer max_number` | Specify the maximum number of Consumers that may be connected to a Channel at a time. If a Consumer tries to connect, while this limit is exceeded, AdminLimitExceeded is raised. Note that this property can also be set programatically via the `set_admin` operation. | int $> 0$ | maximum int value |
| `max_events_ per_consumer` | Specifies how many Events a ProxySupplier at most should queue for a consumer. If this number is exceeded Events are discarded according to the DiscardPolicy configured for the ProxySupplier. | int $> 0$ | 100 |
| `max_batch_size` | Specifies the maximal number of Messages a SequencePushSupplier should queue before a delivery to its connected SequencedPushConsumer is forced. | int $>= 0$ | 1 |

Table 20.1: Notification Service Properties

| Property | Description | Type | Default |
|---|---|---|---|
| `order_policy` | Specify how events that are queued should be ordered. Valid values are:<br>• AnyOrder<br>• PriorityOrder<br>• DeadlineOrder<br>• FifoOrder | string | Priority-Order |
| `discard_policy` | Specifies which Events are discarded if more than the maximal number of events are queued for a consumer. Valid values are:<br>• AnyOrder<br>• PriorityOrder<br>• DeadlineOrder<br>• FifoOrder<br>• LifoOrder | string | Priority-Order |
| `consumer. backout_interval` | After a delivery to a Consumer has failed the Channel will pause delivery to that Consumer for a while before retrying. This property specifies how long a consumer should stay disabled. | milli-seconds | 1000 |
| `consumer. error_threshold` | Each failed delivery to a consumer increments an errorcounter. If this errorcounter exceeds the specified value the consumer is disconnected from the channel. | int $>= 0$ | 3 |
| `default_filter_factory` | Specify which FilterFactory (`CosNotifyFilter::FilterFactory`) the attribute `EventChannel::-default_filter_factory` should be set to. Default value is *builtin*. This special value implies that a FilterFactory will be created during start of the EventChannel. Its possible to set this property to a URL that points to another `CosNotifyFilter::FilterFactory` object. In this case no FilterFactory is started by the EventChannel. The URL is resolved by a call to `ORB::string_to_object`. | URL | builtin |

Table 20.1: Notification Service Properties

| Property | Description | Type | Default |
|---|---|---|---|
| `proxy.destroy_causes_disconnect` | Specify if a destroyed Proxy should call the disconnect operation of its consumer/supplier. | boolean | on |

### 20.5.1 Setting up Bidirectional GIOP

If you have set the ORBInitializer property as described in Section 13.1.1 the Notification will automatically configure its POA to use Bidirectional GIOP.

## 20.6 Monitoring the Notification Service

The JacORB Notification Service provides JMX MBean interfaces which make it possible to monitor and control the Service using a JMX console. This section will describe how to start a JMX enabled Notification Service and how to configure your JMX console to access the exposed MBeans.

### 20.6.1 Download MX4J

MX4J is an Open Source implementation of the JMX specification. You need to download the current release of MX4J (currently 3.0.1) from the JMX project page mx4j.sourceforge.net and install MX4J in an appropriate place.

### 20.6.2 Edit Java Service Wrapper configuration

The configuration file *NotifyService-Wrapper-MX4J.conf.template* contains the necessary settings to start the JMX enabled Notification Service. You have to edit the classpath. Replace the token @MX4J_HOME@ with the concrete path to your MX4J installation.

```
# Java Classpath (include wrapper.jar)  Add class path elements as
#  needed starting from 1
wrapper.java.classpath.1=../lib/wrapper-3.x.y.jar
...
wrapper.java.classpath.8=@MX4J_HOME@/lib/mx4j.jar
wrapper.java.classpath.9=@MX4J_HOME@/lib/mx4j-remote.jar
wrapper.java.classpath.10=@MX4J_HOME@/lib/mx4j-tools.jar
```

After customization of the configuration file it must be renamed to *NotifyService-Wrapper.conf* as the start script reads its configuration from the so called file.

### 20.6.3 Start the Service

consult section 20.3.1 for further details to start the Service.

### 20.6.4 Connecting to the management console

The Notification Service web management console will be available at http://localhost:8001.

A JSR 160 RMI connector allows a management console to connect to the Notification Service. The Service URL is `service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jndi/COSNotification`.

## 20.7 Extending the JacORB Notification Service

### 20.7.1 Adding custom Filters

The JacORB Notification Service supports the full ETCL filter grammar. If you need to use your own specialized filters you have to provide an implementation of the Filter Interface. You can extend the class `org/jacorb/notification/filter/AbstractFilter.java` that provides generic filter management. See the class `org/jacorb/notification/filter/bsh/BSHFilter.java` for an example of an Beanshell based custom filter. Additionally to the standard ETCL grammar the FilterFactory will try to load all filter grammars specified in jacorb.properties. An entry must have the following form: `jacorb.notification.filter.plugin.GRAMMAR=CLASSNAME` where *GRAMMAR* should be replaced by the name of the grammar and *CLASSNAME* should be replaced by the classname of the custom filter implementation.

custom filters can then be created by invoking `FilterFactory::create_filter` or `FilterFactory::create_mapping_filter` and specifying *GRAMMAR* as parameter.

# 21 Using Java management Extentions (JMX)

This section describes how to use the Java Management Extention API along with JacORB to instrument both the orb and application that use JacORB.

## 21.1 MX4J and JMX over IIOP

This section describes how to instrument a JacORB application using the MX4J JMX implementation. MX4J is an open source JMX implementation available at http://mx4j.sourceforge.net. This section also shows how to use JMX over IIOP. This allows JMX to use an existing JacORB ORB for RMI communications and the JacORB Naming Service to register you JMX MBeanServer.

To setup the JVM environment, three system defines are neccessary:

```
-Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
-Djava.naming.provider.url=corbaloc:iiop:localhost:9101/StandardNS/NameServer-POA/_root
-Djavax.rmi.CORBA.PortableRemoteObjectClass=org.jacorb.orb.rmi.PortableRemoteObjectDelegateImpl
```

The first system property tells the Java JNDI subsystem to use the CORBA Naming Service for its naming repository. The second property is a pointer to the JacORB Naming Service instance. The third property tells the Java Remote object system to use JacORB's Portable Remote Object implementation. This is required so that JacORB can associate an RMI object with a CORBA object on one of its POAs.

The sample code for creating a MBeanServer is shown below

```
// The MBeanServer to which the JMXConnectorServer will be registered in
jmxServer = MBeanServerFactory.createMBeanServer();

// The address of the connector
HashMap environment = new HashMap();
org.jacorb.orb.rmi.PortableRemoteObjectDelegateImpl.setORB(orb);
JMXServiceURL address = new JMXServiceURL("service:jmx:iiop://localhost/jndi/jmxSnmpTrapNotify");
JMXConnectorServer cntorServer = JMXConnectorServerFactory.newJMXConnectorServer(address,

// Add MBeans
jmxServer.registerMBean(trapReceiver, new ObjectName("TrapReceiver:counts=default"));

// Start the JMXConnectorServer
cntorServer.start();
```

The first line creates the MBeanServer. The next 4 lines creates the remote JMX connection. The "'setORB()"' call assignes a previously initialized ORB to the Remote Object delegate. All RMI over

IIOP communications will occure via this ORB. The ”‘address”’ is the name of the MBeanServer as known in the Naming service. The portion after ”‘/jndi/”’ is the Naming Service name. The next line registers a MBean with the MBeanServer. The last line starts the MBeanServer.

A JMX console may then be used to monitor the JacORB application. For example, MC4J (http://mc4j.sourceforge.net) may be used. When setting up a mc4j connection, use the connection type JSR160 and set the server URL to the name as registered in the JacORB naming service, such as ”‘service:jmx:iiop://localhost/jndi/jmxSnmpTrapNotify”’.

# 22 JacORB Utilities

In this chapter we briefly explain the executables that come with JacORB. These include the IDL-compiler, a utility to decode IORs and print their components, the JacORB name server, a utility to test a remote object's liveness, etc.

## 22.1 idl

The IDL compiler parses IDL files and maps type definitions to Java classes as specified by the OMG IDL/Java language mapping. For example, IDL interfaces are translated into Java interfaces, and typedefs, structs, const declarations etc. are mapped onto corresponding Java classes. Additionally, stubs and skeletons for all interface types in the IDL specification are generated.

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

### Compiler Options

| | |
|---|---|
| -h \| help | print help on compiler options |
| -v \| version | print compiler version information |
| -d dir | root of directory tree for output (default: current directory) |
| -syntax | syntax check only, no code generation |
| -Dx | define preprocessor symbol x with value 1 |
| -Dx=y | define preprocessor symbol x with value y |
| -Idir | set include path for idl files |
| -Usymbol | undefine preprocessor symbol |
| -W [1..4] | debug output level (default is 1) |
| -all | generate code for all IDL files, even included ones (default is off)<br>If you want to make sure that for a given IDL no code will<br>be generated even if this option is set, use the (proprietary) preprocessor<br>directive `#pragma inhibit_code_generation`. |
| -forceOverwrite | generate Java code even if the IDL files have not<br>changed since the last compiler run (default is off) |
| -ami_callback | generate AMI reply handlers and sendc methods (default is off). See chapter 15 |
| -ami_polling | generate AMI poller and sendp methods (default is off). See chapter 15 |
| -addbackend classname | classname as code generator |
| -backend classname | use classname as compiler (code generator) backend. |

If no generator is specified then it will default to simple file output.
Custom generators must implement the interface
`org.jacorb.idl.IDLTreeVisitor`

-i2jpackage x:a.b.c — replace IDL package name x by a.b.c in generated Java code (e.g. CORBA:org.omg.CORBA)

-i2jpackagefile filename — replace IDL package names using list from ¡filename¿. Format as above.

-ir — generate extra information required by the JacORB Interface Repository (One extra file for each IDL module, and another additional file per IDL interface.) (default is off)

-cldc10 — Generate J2ME/CLDC1.0 compliant stubs

-genEnhanced — Generate stubs with toString/equals (only StructType)

-nofinal — generated Java code will contain no final class definitions, which is the default to allow for compiler optimizations.

-unchecked_narrow — use unchecked_narrow in generated code for IOR parameters in operations (default is off). Generated helper classes contain marshalling code which, by default, will try to narrow any object references to statically known interface type. This may involve remote invocations to test a remote object's type, thus incurring runtime overhead to achieve static type safety. The -unchecked_narrow option generates code that will not by statically type safe, but avoids remote tests of an object's type. If the type is not as expected, clients will experience CORBA.BAD_OPERATION exceptions at invocation time.

-noskel — disables generation of POA skeletons (e.g., for client-side use)

-nostub — disables generation of client stubs (for server-side use)

-diistub — generate DII-based client stubs (default is off)

-sloppy_forward — allow forward declarations without later definitions (useful only for separate compilation).

-sloppy_names — less strict checking of module name scoping (default: off) CORBA IDL has a number of name resolution rules that are stricter than necessary for Java (e.g., a struct member's name identifier must not equal the type name). The -sloppy_names option relaxes checking of these rules. Note that IDL accepted with this option will be rejected by other, conformant IDL compilers!

-sloppy_identifiers — permit illegal identifiers that differ in case (04-03-12:3.3.2) (default: off)

-permissive_rmic — tolerate dubious and buggy IDL generated by JDK's rmic stub generator (e.g., incorrectly empty inheritance clauses), includes -sloppy_names.

-generate_helper *compatibilty* — controls the compatibilty level of the generated helper code. Valid values are:
**deprecated** uses CORBA 2.3 API. this API version is part of the JDK.
**portable** uses CORBA 2.4 API. the usage of this option mandates the use of the JacORB provided *org.omg.** classes on the bootclasspath. This is the default.
**jacorb** uses JacORB API. The generated helper code will contain references to JacORB classes. The helpers will use the CORBA 2.4 API but won't be portable anymore. There's no need to put the *org.omg.** classes provided by JacORB

on the bootclasspath.

## i2jpackage

The `-i2jpackage` switch can be used to flexibly redirect generated Java classes into packages. 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. It is also possible to specify a file containing these mappings using the `-i2jpackagefile` switch.

### Example 1

given the following IDL definition

```
struct MyStruct
{
    long value;
};
```

Invoking idl without the i2jpackage option will generate (along with other files) the java file MyStruct.java

```
/**
 * Generated from IDL struct "MyStruct".
 *
 * @author JacORB IDL compiler V 2.3, 18-Aug-2006
 * @version generated at 07.12.2006 11:46:28
 */

public final class MyStruct
        implements org.omg.CORBA.portable.IDLEntity
{
    [...]
}
```

Note that the class does not contain a package definition.

The option -i2jpackage :com.acme will place any identifier without scope into the java package com.acme. Thus we get:

```
package com.acme;

/**
```

```
 * Generated from IDL struct "MyStruct".
 *
 * @author JacORB IDL compiler V 2.3, 18-Aug-2006
 * @version generated at 07.12.2006 11:46:28
 */

public final class MyStruct
        implements org.omg.CORBA.portable.IDLEntity
{
    [...]
}
```

### Example 2

```
module outer
{
    struct OuterStruct
    {
        long value;
    };

    module inner
    {
        struct InnerStruct
        {
            long value;
        };
    };
};
```

If you're not using the i2jpackage option, the IDL compiler will generate the classes *outer.OuterStruct* and *outer.inner.InnerStruct*.

Again using the i2jpackage it's possible to map IDL modules to different java packages. `$ idl -i2jpackage outer:com.acme.outer` will generate the classes *com.acme.outer.OuterStruct* and *com.acme.outer.inner.InnerStruct*.

`$ idl -idjpackage inner:com.acme.inner` will generate the classes *outer.OuterStruct* and *outer.com.acme.inner.InnerStruct*.

Note: See Section 10.4 if you intend to use the i2jpackage option in conjunction with the JacORB IfR and are using #pragma prefix statements in your IDL.

## Compiler Options

If one is building from Ant it is possible to invoke the compiler directly using the supplied Ant task, JacIDL. To add the taskdef add the following to the ant script:

```
<taskdef name="jacidl" classname="org.jacorb.idl.JacIDL"/>
```

The task supports all of the options of the IDL compiler.

Table 22.1: JacIDL Configuration

| Attribute | Description | Required | Default |
|---|---|---|---|
| srcdir | Location of the IDL files | Yes | |
| destdir | Location of the generated java files | Yes | |
| includes | Comma-separated list of patterns of files that must be included; all files are included when omitted. | No | |
| includesfile | The name of a file that contains include patterns. | No | |
| excludes | Comma-separated list of patterns of files that must be excluded; files are excluded when omitted. | No | |
| excludesfile | The name of a file that contains include patterns. | No | |
| defaultexcludes | Indicates whether default excludes should be used (yes — no); default excludes are used when omitted. | No | |
| includepath | The path the idl compiler will use to search for included files. | No | |
| parseonly | Only perform syntax check without generating code. | No | False |
| noskel | Disables generation of POA skeletons | No | False |
| nostub | Disables generation of client stubs | No | False |
| diistub | Generate DII-based client stubs | No | False |
| sloppyforward | Allow forward declarations without later definitions | No | False |
| sloppynames | Less strict checking of names for backward compatibility | No | False |
| generateir | Generate information required by the Interface Repository | No | False |
| all | Generate code for all IDL files, even included ones | No | False |
| nofinal | Generate class definitions that are not final | No | False |
| forceoverwrite | Generate code even if IDL has not changed. | No | False |
| uncheckedNarrow | Use unchecked_narrow in generated code for IOR parameters in operations. | No | False |
| ami | Generate ami callbacks. | No | False |
| debuglevel | Set the debug level from 0 to 4. | No | 0 |
| helpercompat | control the portability of the generated helper code. | No | portable |

### Nested Elements

Several elements may be specified as nested elements. These are `<define>`, `<undefine>`, `<include>`, `<exclude>`, `<patternset>` and `<i2jpackage>`. The format of `<i2jpackage>` is `<i2jpackage names="x:y">`

### Examples

The task command

```
<jacidl destdir="${generate}"
        srcdir="${idl}"
/>
```

compiles all *.idl files under the $idl directory and stores the .java files in the $generate directory.

```
<jacidl destdir="${generate}" srcdir="${idl}">
   <define key="GIOP_1_1" value="1"/>
</jacidl>
```

like above, but additionaly defines the symbol GIOP_1_1 and sets its (optional) value to 1.

```
<jacidl destdir="${generate}"
        srcdir="${idl}"
        excludes="**/*foo.idl"
/>
```

like the first example, but exclude all files which end with foo.idl.

## 22.2 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 [**?**]. 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.

## 22.3 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
```

## 22.4 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.

## 22.5 dior

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

### Usage

```
$ dior [-u] [-c] -i <IOR-string> | -f <filename>
```

- Option '-u' Decodes and prints the object key.
- Option '-c' Decodes and prints a corbaloc representation of the objectkey.

### 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
```

## 22.6 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 -i <IOR-string> | -f <filename>
```

## 22.7 ir

This command starts the JacORB Interface Repository, which is explained in chapter 10.

### Usage

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

## 22.8 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>
```

## 22.9 ks

This command starts the JacORB KeyStoreManager, which is explained in chapter 11

### Usage

```
$ ks
```

## 22.10 fixior

This command patches host and port information into an IOR file.

### Usage

```
$ fixior <host> <port> <ior_file>
```

# 23 Transport Current

Using the org.jacorb.transport.Current Feature

*by Iliyan Jeliazkov*

## 23.1 Scope and Context

There is no standard-mandated mechanism to facilitate obtaining statistical or pretty much any operational information about the network transport which the ORB is using. While this is a direct corollary of the CORBA's design paradigm which mandates hiding all this hairy stuff behind non-transparent abstractions, it also precludes effective ORB and network monitoring.

The Transport::Current feature intends to fill this gap by defining a framework for developing a wide range of solutions to this problem. It also provides a basic implementation for the most common case - the IIOP transport.

By definition, transport-specific information is available in contexts where the ORB has selected a Transport:

- Within Client-side interception points;
- Within Server-side interception points;
- Inside a Servant up-call

The implementation is based on a generic service-oriented framework, implementing the Transport::Current interface. It is an optional service, which can be dynamically l oaded. This service makes the Transport::Current interface available through orb-¿resolve_initial_references() . The basic idea is simple - whenever a Transport is chosen by the ORB, the Transport::Current (or a protocolo-specific derivative) will have access to that instance and be able to provide some useful information.

## 23.2 Programmer's Reference

Consider the following IDL interface to access transport-specific data.

```
module org
{
```

```
  module jacorb
  {

    module transport
    {
      /// A type used to represent counters
      typedef unsigned long long CounterT;

      // Used to signal that a call was made outside the
      // correct invocation context.

      exception NoContext
      {
      };

      // The main interface, providing access to the Transport-specific
      // information (traits), available to the current thread of
      // execution.

      local interface Current
      {
          /// Transport ID, unique within the process.
        long id() raises (NoContext);

          /// Bytes sent/received through the transport.
          CounterT bytes_sent() raises (NoContext);
          CounterT bytes_received() raises (NoContext);

          /// Messages (requests and replies) sent/received using the current
          /// protocol.
          CounterT messages_sent() raises (NoContext);
          CounterT messages_received() raises (NoContext);

          /// The absolute time (miliseconds) since the transport has been
          /// open.
          TimeBase::TimeT open_since() raises (NoContext);
      };

    };

  };

};
```

As an example of a specialized Transport::Current is the Transport::IIOP::Current, which derives from Transport::Current and has an interface, described in the following IDL:

```
module org
{

  module jacorb
  {

    module transport
    {
      /// A type used to represent counters
      typedef unsigned long long CounterT;

      // Used to signal that a call was made outside the
      // correct invocation context.

      exception NoContext
      {
      };

      // The main interface, providing access to the Transport-specific
      // information (traits), available to the current thread of
      // execution.

      local interface Current
      {
          /// Transport ID, unique within the process.
        long id() raises (NoContext);

          /// Bytes sent/received through the transport.
          CounterT bytes_sent() raises (NoContext);
          CounterT bytes_received() raises (NoContext);

          /// Messages (requests and replies) sent/received using the current
          /// protocol.
          CounterT messages_sent() raises (NoContext);
          CounterT messages_received() raises (NoContext);

          /// The absolute time (miliseconds) since the transport has been
          /// open.
          TimeBase::TimeT open_since() raises (NoContext);
      };

    };
```

```
  };

};
```

## 23.3 User's Guide

The org.jacorb.transport.Current can be used as a base interface for a more specialized interfaces. Iowever, it is not required that a more specialized Current inherits from it.

Typical, generic usage is shown in the tests/regression/src/org/jacorb/test/transport/IIOPTester.java test:

```
...
   // Get the Current object.
   Object tcobject =
      orb.resolve_initial_references ("JacOrbIIOPTransportCurrent");
   Current tc = CurrentHelper.narrow (tcobject);

   logger.info("TC: ["+tc.id()+"] from="+tc.local_host() +":"
               + tc.local_port() +", to="
               +tc.remote_host()+":"+tc.remote_port());

   logger.info("TC: ["+tc.id()+"] sent="+tc.messages_sent ()
               + "("+tc.bytes_sent ()+")"
               + ", received="+tc.messages_received ()
               + "("+tc.bytes_received ()+")");
...
```

### 23.3.1 Configuration, Bootstrap, Initialization and Operation

To use the Transport Current features the framework must be loaded through the Service Configuration framework. For example, using something like this:

```
...
        Properties serverProps = new Properties();

        // We need the TC functionality
        serverProps.put ("org.omg.PortableInterceptor.ORBInitializerClass.
                             server_transport_current_interceptor",
                         "org.jacorb.transport.TransportCurrentInitializer");

        serverProps.put ("org.omg.PortableInterceptor.ORBInitializerClass.
                             server_transport_current_iiop_interceptor",
```

```
                        "org.jacorb.transport.IIOPTransportCurrentInitializer");

        serverORB = ORB.init(new String[0], serverProps);
...
```

The ORB initializer registers the ”JacORbIIOPTransportCurrent” name with the orb, so that it could be resolved via orb-¿resolve_initial_references(”JacORbIIOPTransportCurrent”).

Note that any number of transport-specific Current interfaces may be available at any one time.

# 24 JacORB Threads

Threads that are created and used by JacORB are described below.

## Long–lived threads

### RequestProcessor

The RequestProcessor thread invokes servant code when the thread is assigned a request from the RequestController. This thread invokes firstly the server request interceptors, then the servant manager, and then the servant code. Finally, the RequestProcessor invokes interceptors and servant managers and writes results to the socket when the servant returns the control flow.

The number of RequestProcessor threads which can run is between `jacorb.poa.thread_pool_min` and `jacorb.poa.thread_pool_max` times the number of POAs, or just between those two bounds when `jacorb.poa.thread_pool_shared` is set to "on". RequestProcessor threads will terminate when the POA is destroyed (in other words when the property is set to "off" and when every POA has it's own pool of RequestProcessors) or when `ORB.shutdown()` is called, subject to the value of the `jacorb.poa.thread_pool_shared` property.

The RequestProcessor thread is implemented in `org/jacorb/poa/RequestProcessor.java`. Thread instances are pooled in `org/jacorb/poa/RPPoolManager.java`.

### RequestController

The RequestController assigns requests to RequestProcessors and keeps track of active requests, object and POA state. The POA state is checked when the ServerMessageReceptor reads a request from the socket. Request processing can continue if the POA state is active. However, if the POA is inactive or if it is being shut down, then the request is rejected. If the target object is present and not being deactivated, then a RequestProcessor thread is allocated from the pool and the request is handed over to the that thread. One RequestController thread is always provided for each POA: the thread is terminated when the POA is destroyed.

The RequestController thread is implemented in `org/jacorb/poa/RequestController.java`. A reference to the thread is retained by `org/jacorb/poa/POA.java`.

## ServerSocketListener, SSLServerSocketListener

These two threads listen on their respective server sockets and accept new connections. Accepted connections are handed to a thread pool. The ServerMessageReceptor uses the thread pool to listen on connections for individual messages.

There can be a maximum of one ServerSocketListener and one SSLServerSocketListener per ORB, depending on the configuration. These threads will terminate when `ORB.shutdown()` is called.

The ServerSocketListener and SSLServerSocketListener threads are implemented in the inner classes `Acceptor` and `SSLAcceptor` in `org/jacorb/ orb/iiop/IIOPListener.java`: a reference is retained by the class.

## ServerMessageReceptor

ServerMessageReceptor threads listen on established connections and read new requests from them. The request's message header is decoded and the POA name is retrieved from the object key after basic checks are made. The request is then handed to the POA for scheduling by the RequestController.

The number of ServerMessageReceptor threads is between 0 and the value of `jacorb.connection.server .max_receptor_threads`. This upper bound also indicates the maximum number of connections that can be serviced simultaneously. The maximum number of idle threads can be configured using `jacorb.connection.server.max_idle_receptor_threads`.

ServerMessageReceptor threads terminate when either `ORB.shutdown()` is called or when the number of idle threads exceeds the maximum specified by `jacorb.connection.server.max_idle_receptor_threads`.

The ServerMessageReceptor thread is implemented in `org/jacorb/orb/giop/MessageReceptor.java`: instances are pooled in `org/jacorb/orb/giop/MessageReceptorPool.java`. Both these classes rely on and implement interfaces from JacORB's generic thread pool in `org/jacorb/util/threadpool`.

## ClientMessageReceptor

ClientMessageReceptor threads listen on established connections and read new replies recieved from them. The request's message header is decoded and the reply is handed back to the thread that sent the original request after basic checks are performed. The number of threads which are allowed is between 0 and the value of `jacorb.connection.client .max_receptor_threads`. This upper bound also indicates the maximum number of connections that can be serviced simultaneously. The maximum number of idle threads allowed can be set using `jacorb.connection.client.max_idle_receptor_threads`.

ClientMessageReceptor threads terminate when either `ORB.shutdown()` is called or when the number of idle threads exceeds the maximum specified by `jacorb.connection.client.max_idle_receptor_threads`.

This thread is implemented in `org/jacorb/orb/giop/MessageReceptor.java` and its instances are pooled in `org/jacorb/orb/giop/MessageReceptorPool.java`. Both these classes rely on and implement interfaces from JacORB's generic thread pool in `org/jacorb/util/threadpool`.

### BufferManagerReaper

The BufferManagerReaper thread ensures that the extra-large buffer cache entry will not live longer than the time specified by `jacorb.bufferManagerMaxFlush`. The BufferManagerReaper thread exits when `ORB.shutdown()` is called.

This thread is implemented as inner class `Reaper` in `org/jacorb/orb/BufferManager.java` and a reference is kept by the class.

### AOMRemoval

This thread is used to ensure that calls to deactivate_object return immediately. When an object is removed it is placed on a java.util.concurrent.LinkedBlockingQueue which this thread processes to finish deactivation of the objects.

## Short–lived threads

### POAChangeToActive

The POAChangeToActive thread asynchronously sets the state of those POAs controlled by a POAManager to active. A new thread will be created whenever `POAManager.activate()` is called. The thread terminates when all POAs have been activated.

The POAChangeToActive thread is implemented as an anonymous inner class in `org/jacorb/poa/POAManager.java`.

### POAChangeToInactive

The POAChangeToInactive thread asynchronously sets the state of the POAs controlled by a POAManager to inactive. A new thread will be created whenever `POAManager.deactivate()` is called. The thread terminates when all POAs have been deactivated.

The POAChangeToInactive thread is implemented as an anonymous inner class in `org/jacorb/poa/POAManager.java`.

## POAChangeToDiscarding

The POAChangeToDiscarding thread asynchronously sets the state of those POAs controlled by a POAManager to discarding. A new thread is created whenever `POAManager.discard_requests()` is called. This thread terminates when all POAs have been set to discarding.

The POAChangeToDiscarding thread is implemented as an anonymous inner class in `org/jacorb/poa/POAManager.java`.

## POAChangeToHolding

The POAChangeToHolding thread asynchronously sets the state of those POAs controlled by a POAManager to holding. A new thread is created whenever `POAManager.hold_requests()` is called. This thread when all POAs have been set to holding.

The POAChangeToHolding thread is implemented as an anonymous inner class in `org/jacorb/poa/POAManager.java`.

## POADestructor

The POADestructor thread allows asynchronous destruction of a POA. This thread initially synchronizes with the RequestController which waits until all active requests have been finished. Then, all unprocessed requests are discarded by the RequestController thread and destruction of the POA is completed. The thread will then exit.

One POADestructor thread is created whenever `POA.destroy()` is called. Note that destroying a POA will destroy all child POAs. Accordingly, there will be many threads as there are POAs, including child POAs, which are to be destroyed.

The POADestructor thread is implemented as an anonymous inner class in `org/jacorb/poa/POA.java`.

## PassToTransport

The PassToTransport thread is created and performs the network send task whenever a request is sent with the sync scope set to `SYNC_NONE`. The thread exits when it is finished sending and allows the client thread to return immediately.

The PassToTransport thread is implemented as an anonymous inner class in `org/jacorb/orb/Delegate.java`.

## ReplyReceiverTimer

The ReplyReceiverTimer thread manages the termination point for reply timeouts. The thread is created for each anticipated reply when the ReplyEndTime policy is set. The thread exits when the timeout expires

or the anticipated reply is received before timeout expires.

The ReplyReceiverTimer thread is implemented as inner class `Timer` in `org/jacorb/orb/ReplyReceiver.java` and a reference is kept by the class.

## SocketConnectorThread

The SocketConnectorThread thread connects to the socket for every new connection to the server when `jacorb.connection.client.connect_timeout` is set to a value greater than zero (0). The SocketConnectorThread thread provides timeout control which is not available in older JDK versions

The thread exits when either the connection is successfully established or when the timeout expires.

The ReplyReceiverTimer thread is implemented as an anonymous inner class in `org/jacorb/orb/ClientIIOPConnection.java`.

# 25 Classpath and Classloaders

This chapter explains some of the problems that may be encountered with classpath and classloaders.

## 25.1 Running applications

By default JacORB is shipped with runtime scripts to simplify running an application. These scripts use the Java Endorsed Standards Override Mechanism in order to ensure that the JacORB implementation classes and the supplied OMG classes are found in preference to any bundled within the JVM. This mechanism is documented here http://java.sun.com/j2se/1.5.0/docs/guide/standards

The mechanism utilises the Xbootclasspath to place the classes first. If this is not used then the Sun OMG classes may be found first. The issue that may be encountered here is if JacORB is released with newer versions of the OMG classes than is distributed within the JVM. Therefore the JacORB classes should be used in preference.

### 25.1.1 ORBSingleton

Unlike an ORB.init(args,props) where a developer may pass arguments initialising an ORBSingleton with ORB.init() does not. This means that unless the developer has either

- Started the JVM supplying ORBSingletonClass and ORBClass properties
- Overridden System properties prior to calling ORBInit with ORBSingletonClass and ORBClass properties

the OMG ORB class will initialise the wrong ORBSingleton **if** endorsed directories are **not** being used. If endorsed directories are being used the JacORB OMG ORB class will automatically load the correct Singleton.

## 25.2 Interaction with Classloaders

The endorsed directory mechanism means that the JacORB classes will be loaded into the bootstrap classloader. If the developer has chosen to instantiate their own child classloader and load the JacORB classes within this (e.g. via URLClassLoader downloading the classes over the network) several problems may be encountered:

### Garbage Collection

The Sun JVM will load its OMG ORB classes in preference to those within the child classloader. This means that it will retain a static link to the JacORB ORBSingleton implementation within the child classloader. Therefore the classes cannot be fully garbage collected once the classloader is disposed of.

### Class Conflict

As described above the Sun OMG ORB class matains a static ORBSingleton reference. If a second class loader is instantiated, as a ORBSingleton already exists in the parent bootclassloader it will not be created. However when the JacORB code checks that

```
ORB.init () instanceof org.jacorb.orb.ORBSingleton
```

it will fail. This is because the ORBSingleton class created in the first classloader is different to the ORBSingleton class created in the second classloader. This behaviour is documented within the Java Language Specification here http://java.sun.com/docs/books/jls/third_edition/html/execution.html#12.1.1 and a paper describing the behaviour may be found here http://www.tedneward.com/files/Papers/JavaStatics/JavaStatics.pdf

### Solving the Problem

The above problem occurs as java.net.URLClassLoader uses the parent-first class-loader delegation model. To solve the issue, the simplest and most effective solution is to use child-first class-loader delegation model. An example of this may be found here http://www.qos.ch/logging/classloader.jsp

This model ensures that parent delegation occurs only **after** an unsuccessful attempt to load a class from the child. Therefore the org.omg.CORBA.* classes supplied with JacORB would be found and used in preference to the OMG classes supplied by Sun in the bootclassloader. The ORBSingleton would be created entirely within the child classloader with no external references. This means the second classloader would also create its own, entirely isolated Singleton class.

# Bibliography