AdironORB Manual

by David Grandinetti, Joncheng Kuo, and Polar Humenn

AdironORB Manual

by David Grandinetti, Joncheng Kuo, and Polar Humenn

Copyright © 2004 Adiron LLC
All rights reserved.

Table of Contents

1. Introduction
    - About This Document 1
    - About AdironORB 1
    - History and Acknowledgements 2

2. Architecture
    - Overview 4
    - ORB Initialization and Configuration 5
    - Pluggable Thread Models 7

3. Compilation and Installation
    - Environment for Building AdironORB 9
    - Downloading AdironORB 9
    - Building AdironORB 9
    - Installing AdironORB 10
    - Using AdironORB 12

4. Configuration
    - Overview 15
    - Ways of Configuring the AdironORB 17
    - AdironORB Parameters 18

5. Proprietary API and Policies
    - Access the Internals of the ORB 21
    - Accessing the Transport Information 22
    - Communicating with Interceptors 24
    - ORB Policies specific to AdironORB 24

6. References

A. AdironORB Error Codes

Chapter 1. Introduction

Table of Contents

About This Document 1
About AdironORB 1
History and Acknowledgements 2

About This Document

This document provides information required to use AdironORB. It describes how to install, configure and use AdironORB. It also explains how to use the IDL compiler.

For any comments or questions about this documentation and AdironORB, please send email to the SL3 Users mailing list `<sl3-users@adiron.com>`. You may subscribe to this list at / http:/greene.case.syr.edu/mailman/listinfo/sl3-users.

This document is the first version of the AdironORB User Guide. It has grown out of the original OpenORB documentation. Also, much of the information regarding the history of OpenORB was giorinally on the Community OpenORB website [http://openorb.sf.net]. It will be completed step by step. Contributions would be greatly appreciated.

About AdironORB

AdironORB is a CORBA Object Request Broker (ORB) developed in Java. It complies with the CORBA/IIOP 2.6[1] specification and provides many features and flexible ways for extensions.

AdironORB has been designed to provide a reliable foundation for distributed applications. It combines all CORBA features with implementation specific extensions, with the aim of being the most powerful and complete CORBA implementation in Java.

CORBA technology is becoming increasingly complex, and its feature set is growing. Many users don't require all features to be present in every ORB deployment. AdironORB is the most complete CORBA implementation but not the biggest! Why? Because AdironORB can be configured to fit only the user's requirements. AdironORB is not monolithic middleware, it is a truly modular ORB.

That means that users can define what mechanisms are needed by their applications, so that AdironORB will be loaded with only those mechanisms. AdironORB contains a kernel that has the ability to load only required parts during bootstrap time.

Thus, AdironORB is the best way to have the best implementation, the most complete solution and the most flexible platform!

The current implementation of AdironORB has the following features:

[1] Bidirectional GIOP is supported in interfaces, but it has not been implemented for security reasons. Other optional features such as Asynchronous Messaging, Firewall, Fault Tolerant CORBA, and Common Secure Interoperability (CSIv2) are not yet implemented.

- Compliant with CORBA 2.6[1]
- Developed in Java
- Compliant with J2SDK 1.4
- Multithreaded ORB with flexible and customizable threading models
- Portable Object Adapter (POA) with the ability to add different object adapters
- IIOP 1.2
- Pluggable Messaging Engine - default to GIOP 1.2
- Pluggable Transport - default to TCP/IP
- Portable Interceptors
- DynAny, DII, DSI
- Code Set support
- IDL compiler - IDL to Java and Java to IDL
- Several tools: IDL to HTML, IDL to RTF, and an IOR parser

History and Acknowledgements

The AdironORB has evolved from the work of several people. The original code was developed under the JavaORB project by Jerome Daniel. The JavaORB project was started in October 1998. The version 1.0 release was in March 1999, the same month in which Jerome founded the Distributed Object Group (DOG). DOG was a collaboration of Jerome Daniels, Xavier Blanc, Nicolas Charpentier, and Vincent Vallee. For more information of the history of the Distributed Object Group, including a more detailed account of members and contributions, go to their web site at http://dog.team.free.fr [http://dog.team.free.fr].

In June 2000, the DOG became part of the Exolab Group (supported by Intalio, Inc.) and it was around this time that Chris Wood joined the developer team. About a month later Marina Daniel also joined the team. While under the guidance of the Exolab Group, several months of in-house development led to the next release. This new release was now under the name OpenORB. More information about the work on OpenORB done by the Exolab Group (as well as the full list of contributors) can be found at http://www.openorb.org [http://www.openorb.org].

By the 4th quarter of 2001, with the end of the dotcom era, Intalio was forced to reconsider thier involvement in the OpenORB project. At this time, a movement was started by several of the community members. The idea was to get Intalio to turn the OpenORB project into a real, community driven, open-source project after the example of many Apache projects. Unfortunately, no agreement was reached with Intalio for doing this. As a result on January 8th 2002, the OpenORB project was forked and established at SourceForge as the Community OpenORB. The new project was founded by Stephen on-McCnell, Michael Rumpf, Shawn Boyce, Jesper Pedersen, J. Scott Evans, and Viacheslav Tararin.

On June 8th, 2002, Intalio announced that they were ending thier support of the OpenORB project in an official capacity. They would still support the developers that forked The Community OpenORB, but

could not host the project on Exolab.

The Community OpenORB project has substantially advanced the source base since thier initial inception and in September 2002 released verion 1.3.0 to the community. The Community OpenORB is still a very active project and there is sure to be substantially more information about the current status of the project at http://openorb.sf.net [http://openorb.sf.net].

In March of 2002, Polar Humenn (of Syracuse University) was looking for an open-source ORB to use as a base for a project. It was decided that OpenORB was the best choice for the needs of the project, but it would be quite difficult to change the transport mechanism that the ORB used. Getting in touch with the Community OpenORB people, Polar found out that Michael Rumpf was working on redesigning the internals of the ORB for the 2.0 release. Among the goals of both Michael and Polar were: separation of the messaging layer from the transport layer, separation of both messaging and transport from the socket/network layer, and restructuring the internal ORB code to use the Apache Avalon Framework.

This release of the Community OpenORB has been renamed AdironORB, as it in the process of transitioning itself as one of the many fine projects at Adiron LLC [http://www.adiron.com]. AdironORB, version 1.0, represents the work done at Syracuse University by Polar Humenn, Joncheng Kuo, and David Grandinetti over the past year. We would like to thank the entire group of developers from the Community OpenORB for their support and help during this time, as well as everyone who has helped get the project to where it is now.

Chapter 2. Architecture

Table of Contents

Overview....4
ORB Initialization and Configuration....5
Pluggable Thread Models....7

AdironORB is a complete redesign of the Community OpenORB that makes the internals of the ORB more modular. The internal structure of the ORB has also been reorganized to fit more with the Apache Avalon Framework design philosophies.

A more modular ORB means that one part can easily be replaced by another one. Thanks to this architecture, it is possible to develop specific parts to fulfill application requirements.

For example, it's possible to replace the inter-ORB protocol by another one. Currently, only the IIOP (GIOP-over-TCP/IP) protocol is available in this distribution. However, it would be very easy to replace that with an SSL based transport mechanism, or even the new SECP protocol. As a demonstration, a GIOP-over-local-communication protocol that uses pipes for the communication between ORBs inside the same JVM process is provided in this distribution.

Overview

One of the approaches in achieving a more modular ORB is to provide a layered architecture. The block diagram shown in this section illustrates the major layers of the AdironORB. The client-side of the ORB sits at the left half of the diagram; the server-side sits at the right half. We will brief the layers in this diagram from top to bottom, left to right.

At the top, the *configuration layer* is in charge of ORB configuration and initialization. We use the Apache Avalon Framework [http://avalon.apache.org] to configure and initialize ORB instances.

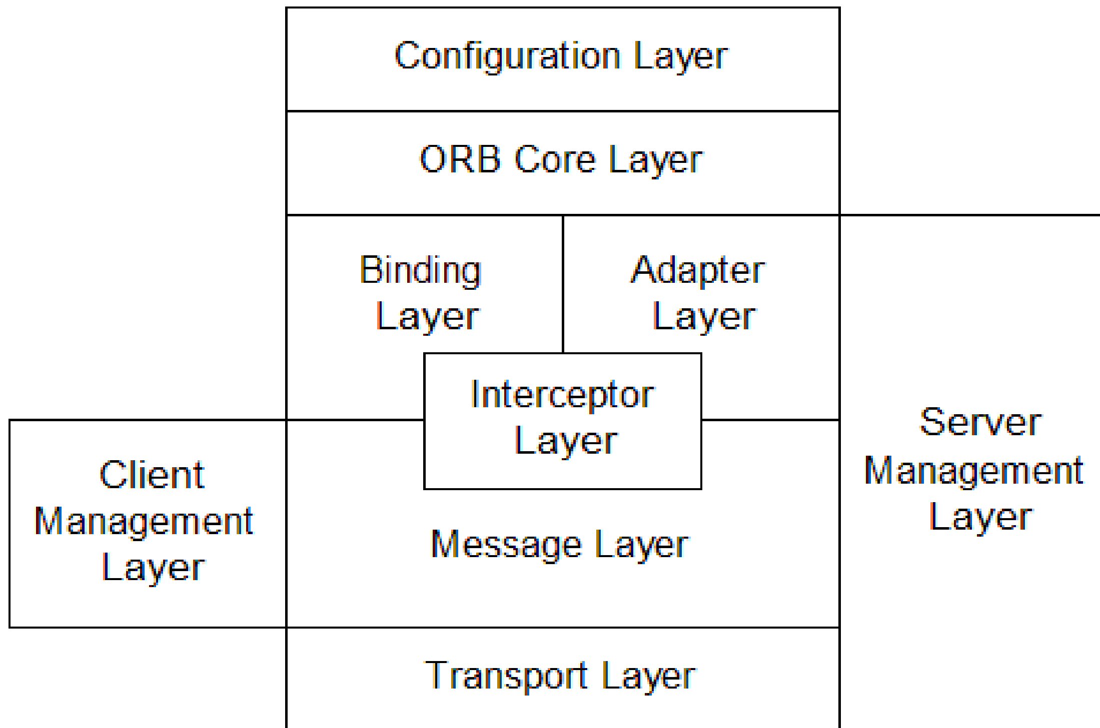
The *ORB core layer* provides the essential functionality of an ORB. It implements the ORB, ORB singleton, object stub, type code, DII, DSI, and Dynamic Any.

The *binding layer* is used for the client-side ORB only. It implements CORBA Delegate and provides associations between Delegates and client channels. These associations are called object reference bindings.

The *adapter layer* is used for the server-side ORB only. It handles the registration and look up of object adapters. AdironORB implements the Portable Object Adapter (POA) as the default object adapter.

The *interceptor layer* handles the interception of object requests and IOR creation. This layer protrudes into other layers such as the binding layer, adapter layer, message layer, and transport layer to provide request and IOR interception. For example, when object requests are created, if request interceptors are used, object requests will be wrapped with intercept requests that handles the interaction with interceptor managers.

The *client management* layer manages client channels and active client requests.



The *message layer* deals with the protocols used for inter-ORB communications. It uses the abstraction of channels and requests. A channel represents a communication link between a client ORB and a server ORB. A channel is called a client channel at the client side and a server channel at the server side. Requests are the atomic communication units over channels. Requests are used to carry CORBA requests for inter-ORB protocol communications. The message layer is pluggable in the sense that all inter-ORB protocols have to be registered to the ORB. Users may register their own protocols. The default inter-ORB protocol supported by AdironORB is Internet Inter-ORB Protocol (IIOP), which uses the GIOP protocol engine and TCP/IP transport.

The *server management* layer manages the resources and activities on a server-side ORB, including managing server channels and dispatching requests. The server management layer also provides management for threads and protocol acceptors.

The *transport layer* handles network communications, including making/accepting connections and sending/receiving data, using the abstraction of sockets provided by the Java API. The transport layer is pluggable in the sense that you can plug in a module for a specific transport protocol when an inter-ORB protocol is plugged in.

ORB Initialization and Configuration

AdironORB uses the Apache Avalon Framework [http://avalon.apache.org] for ORB initialization and configuration. The `ORB` class implements the Avalon lifecycle interfaces. You may instantiate a new ORB and configure the ORB using these lifecycle interfaces. Or you may use the standard `ORB.init` operation to create a new ORB. When you use the `ORB.init` operation to create a new ORB, AdironORB internally uses an *ORB Loader*, which configures and initializes the new ORB instance using the Avalon lifecycle interfaces.

ORB Creation Using the Lifecycle Interfaces

The `ORB` class implements the following Avalon framework's lifecycle interfaces: `LogEnabled`, `Contextualizable`, `Parameterizable`, `Initializable`, and `Disposable`. It is required to invoke the following methods in sequence on a new ORB instance in order to configure and initialize the ORB properly.

1. `enableLogging` [`org.apache.avalon.framework.logger.LogEnabled`]
2. `contextualize` [`org.apache.avalon.framework.context.Contextualizable`]
3. `parameterize` [`org.apache.avalon.framework.parameters.Parameterizable`]
4. `initialize` [`org.apache.avalon.framework.activity.Initializable`]

ORB Creation Using `ORB.init`

When the `ORB.init` operation is used to create a new ORB, the static `ORB.init` operation creates a new ORB instance and invokes one of the two `set_parameters` methods on the new ORB instance.

In the AdironORB implementation, the `set_parameters` methods do the following: First of all, they look for the ORB Loader class name from the properties (and applet) supplied to the `ORB.init` operation. If nothing is found, the default ORB Loader, `AdironORBLoader`, will be used. The `set_parameters` methods then create an instance of the ORB Loader and invoke the ORB Loader. The ORB Loader, acting like a mini Avalon container, uses the Avalon lifecycle interfaces to configure and initialize the new ORB instance.

ORB Configuration

AdironORB uses the *parameters* defined in the Avalon framework to represent the ORB configuration. A parameter, like a property, is a pair of a name and value. The configuration of an ORB is a set of parameters stored in a `Parameters` object. Applications may query the configuration of an ORB using the proprietary methods provided by the `ORB` class. (See Chapter 5 for details.)

The default ORB Loader, `AdironORBLoader`, uses an *ORB Configurator* to derive the configuration of the ORB. The default ORB Configurator, `PropertiesBasedORBConfigurator`, translates the arguments and properties supplied to the `ORB.init` operation, along with some default configuration specified in the `orb.properties` files, into parameters to form the configuration of the ORB. The ORB Configurator stores this configuration in a `Parameters` object and returns this object to the ORB Loader. The ORB Loader then passes this `Parameters` object to the ORB by invoking the `parameterize` method on the ORB.

ORB Initialization

After an ORB instance is contextualized and parameterized, the `initialize` method on the ORB will be invoked. At this stage, the ORB initializes itself into a fully functional ORB in two steps: Firstly, the ORB creates its core components using an *ORB Initializer*. Secondly, the ORB searches and executes the Portable Interceptor ORB Initializers, if any exist.

In the first step of ORB initialization, the ORB looks for the ORB Initializer class name from the configuration of the ORB, i.e., the parameters set through the `parameterize` method. If no class name is found, the default ORB Initializer, `SimpleIIOP_ORBInitializer`, will be used. The ORB then instantiates and executes the ORB Initializer.

An ORB Initializer is in charge of creating the core components of the ORB, including the support for inter-ORB protocols. An ORB Initializer stores these components in two ways: If a component implements the CORBA `Object` interface, the ORB Initializer registers this component to the ORB as an initial reference. If a component is not a CORBA object but is needed by other parts of the ORB, the ORB Initializer adds the component to the ORB as a *context object*. The ORB uses a `Context` object to hold these context objects. Some of these context objects may be made public for application code. Please see the `ORB.getPublicContextObject` in Chapter 5 for details.

In the second step of ORB initialization, the ORB looks for the Portable Interceptor (PI) ORB Initializers from the configuration of the ORB. These PI ORB Initializers are specified as the names/keys of the parameters of the ORB configuration. If the ORB finds any PI ORB Initializers, the ORB instatiates and executes these PI ORB Initializers.

AdironORB adds an extension to the PI ORB Initializers. If a PI ORB Initializer implements the Avalon framework's lifecycle interfaces such as `LogEnabled`, `Contextualizable`, `Parameterizable`, and `Initializable`, the methods of these interfaces will be invoked in sequence before the `pre_init` operation is called. In this way, the context and configuration of the ORB will be passed to these ORB Initializers through the lifecycle interfaces. This extension gives PI ORB Initializers access to the internal objects of the ORB.

Similarly, if a PI ORB Initializer implements the `Disposable` interface, the `dispose` method will be called after the invocation of the `post_init` operation.

Pluggable Thread Models

One of the pluggable parts of the AdironORB architecture is exemplified by the pluggable threading models on the server side of the ORB. When a server enabled ORB is started, there is a thread manager object declared in the parameters. This thread manager object must implement the `com.adiron.orb.srvmgt.ThreadManager` interface. There are five Thread Manager objects provided with the AdironORB distribution. It would be easy to implement another and use it at runtime.

Here are the five available thread manager classes: (with prefix "com.adiron.orb.srvmgt.thread." omitted)

- `SingleThreadedTM`: This model uses a single thread for everything, including accepting and serving client connections, as well as dispatching client requests. Servers using this model can serve only one client at a time, but this model has the lowest performance cost.
- `NewThreadPerClientTM`: This model creates a new thread for every client connection. This thread is in charge of dispatching client requests as well. Servers using this model can take multiple client connections, but do not allow concurrent requests from the same client.
- `NewThreadPerRequestTM`: This model creates a new thread for every client connection and every request. Servers using this model allow concurrent requests from each client, but this model has a high performance cost.
- `ThreadPoolClientTM`: This model uses a thread pool for serving client connections. The thread serving a client connection is also responsible for dispatching requests from the same connection. Servers using this model are similar to those using the NewThreadPerClientTM model, but this model has better performance because threads are reused.

- `ThreadPoolRequestTM`: This model uses two thread pools: one for serving client connections and the other for dispatching requests. Servers using this model are similar to those using the NewThreadPerRequestTM model, but this model has better performance because threads are reused.

Chapter 3. Compilation and Installation

Table of Contents

Environment for Building AdironORB 9
Downloading AdironORB 9
Building AdironORB 9
Installing AdironORB 10
Using AdironORB 12

This chapter describes the environment and procedures to build and install the AdironORB.

Environment for Building AdironORB

You must have Java 2 SDK, Standard Edition, Version 1.4 (or higher) to build AdironORB. After you install your JDK on your system, you have to set the `JAVA_HOME` environment variable to your JDK installation (e.g., `/usr/local/j2sdk1.4.1` on Unix or `C:\j2sdk1.4.1` on Windows.

Downloading AdironORB

The AdironORB project has many packages, including the **Tools** package, the **AdironORB** package, and Common Object Services packages. Each package is provided in two formats: a gzipped tar file or a zip file. You may download these packages from the AdironORB web site.

For building the AdironORB, you have to download the source distribution of both the **Tools** and **AdironORB** packages. For example, if you choose to download the gzipped tar files, you need the following two files: `Tools-src-X.Y.Z.tgz` and `AdironORB-src-X.Y.Z.tgz` where `X.Y.Z` is the AdironORB version number.

After you download these packages, you have to unpack them to the *same* directory. Each package will be in its own subdirectory.

In some cases, you may download a big tar file that contains all of the AdironORB packages. This file is named like `AdironORB-all-src-X.Y.Z.tgz`.

Building AdironORB

The source distribution of each package has a build script that uses the Apache Ant [http://avalon.apache.org] build tool. This tool is included in the source distribution of the **Tools** package. Thus, you do not have to install Ant. If you use OS/X, you have to edit the `global.properties` file in the **Tools** package to set your Java runtime.

To build a package, you have to invoke the build script corresponding to your operating system in each package: `build.sh` for Unix or `build.bat` for Windows. If you have Cygwin installed on Windows, you may follow the instructions for Unix.

Although building the **AdironORB** package requires building the **Tools** package first, for your convenience, when you invoke the build script of the **AdironORB** package without specifying a target, the build script will first build the **Tools** package by default. Thus, you are not required to invoke the build script of the **Tools** package manually.

For example, if you are on a Unix machine or a Windows machine with Cygwin installed, you may execute the following commands to build AdironORB:

```
cd AdironORB
./build.sh
```

If you are on a Windows machine without CygWin, you may use the following commands instead:

```
cd AdironORB
build
```

If you download the AdironORB as a big tar file that contains all of the AdironORB packages, you may run **./build.sh** (on Unix) or **build.bat** (on Windows) from the top level directory to build every package.

```
    ./build.sh        (on Unix/Cygwin)
or
    build             (on Windows)
```

After building the AdironORB, you will find the generated jar files in the `lib` subdirectory of a package. The name of these jar files are prefixed with "`adironorb`" and postfixed with the version number of AdironORB.

You may also invoke the build script with a target specified. The build script provides many targets such as `jar-all`, `test`, `doc`, `install`, etc. You may invoke the build script with the `help` argument to see a list of supported targets.

Building the Documentation

After building the AdironORB jar files, it is optional to build the documentation. If you choose not to build the documentation, you will not have the documentation when you install AdironORB.

To build the documentation of AdironORB, run the build script with the `doc` target. The generated HTML and PDF documentation will be placed in the `doc` subdirectory.

```
    ./build.sh doc        (on Unix/Cygwin)
or
    build doc             (on Windows)
```

Installing AdironORB

Before you can install AdironORB, you have to edit the `global.properties` file in the **Tools** package to set the `install.path` property. The value of this property must specify the directory you would like AdironORB to be installed in. For example, you may set

```
    install.path=/opt/AdironORB-1.0.0          on Unix
or
    install.path=C:\\opt\\AdironORB-1.0.0      on Windows.
```

Note that you need a double backslash '\\' for each single backslash '\' on Windows.

After modifying the `global.properties` file, you may invoke the build script again with the `in-`

`stall` target for installing AdironORB.

```
    ./build.sh install    (on Unix/Cygwin)
or
    build install         (on Windows)
```

The build script will install all of the necessary files to the directory specified by the `install.path` property. These files include:

- all jar files created during building,
- all external jar files that are needed for the AdironORB runtime,
- configuration files,
- Javadoc and ORB documentation,
- IDL files,
- test and example programs, and
- a `/bin` directory that contains some scripts for running the IDL compiler, an IOR dumper, and the Java to IDL compiler (used for Java-RMI) from the command line.

Here is a simple summary of each jar file produced from compiling the AdironORB source code. This may be helpful to those who want to deploy and AdironORB application without distributing the full AdironORB distribution. The AdironORB jar files consist of:

- `adironorb-omg-X.Y.Z.jar`: the CORBA classes defined by OMG.
- `adironorb-orb-X.Y.Z.jar`: the ORB implementation.
- `adironorb-compiler-X.Y.Z.jar`: the IDL compiler.
- `adironorb-tools-X.Y.Z.jar`: the runtime tools used by the ORB implementation and IDL compiler.

Note

X.Y.Z is the AdironORB version number.

The first three jar files, `adironorb-omg-X.Y.Z.jar`, `adironorb-orb-X.Y.Z.jar`, and `adironorb-tools-X.Y.Z.jar`, must be placed in your class path for running CORBA applications. The first and second jar files contain the core of AdironORB and CORBA runtime, including all the client and server side code. The third jar file conatains small utility classes that almost all AdironORB modules take advantage of.

The last jar file, `adironorb-compiler-X-X-X.jar`, must be installed for building CORBA applications. It is dependent on the `adironorb-tools-X.Y.Z.jar` file, which must be installed in the same class path as the compiler jar file.

In addition to the above AdironORB jar files, there are several support jar files that must be in the ssclapath as well. These jar files are included in the AdironORB installation as well.

- `avalon-framework.jar`: the Apache Avalon Framework.
- `logkit.jar`: the Apache LogKit component.

For information on how to add jar files to the class path, please see your JDK documentation.

Using AdironORB

You may use AdironORB in two ways: The first way is to use AdironORB in a standard CORBA application; the second way is to use AdironORB as an Avalon component. When you run a standard CORBA application, the `ORB.init` operation will be used to create a new ORB. You have to configure certain properties such that the AdironORB's ORB implementation will be used. If you use the AdironORB as an Avalon component, you may instantiate a new ORB instance and configure the ORB using the Avalon lifecycle interfaces.

Using AdironORB in a CORBA Application

A CORBA applet or application uses the `ORB.init` operation to create a new ORB. This operation checks for the following properties to determine the ORB class to use:

- `org.omg.CORBA.ORBClass`: The value of this property identifies the class name of the ORB class.
- `org.omg.CORBA.ORBSingletonClass`: The value of this property identifies the class name of the ORB singleton class.

The `ORB.init` operation searches the above two properties in the following order:

1. Check in the Applet parameter, if any.
2. Check in the `properties` argument, if any.
3. Check in the system properties.
4. Check in the `orb.properties` file, if exists.
5. Fall back on the default ORB class.

Thus, to run a CORBA application using AdironORB, there are four different approaches: If you want to have AdironORB as the default ORB for all CORBA applications on one host machine, then you need to modify (or create, if none exists) the `orb.properties` file in the `lib` subdirectory of your Java runtime. If you, instead, want AdironORB to be the default ORB used for a single user on a host machine, you can place the `orb.properties` file in the user's home directory. If you want to specify the ORB class at runtime, you may use system properties. You may also specify properties in the application and and pass these properties to the `ORB.init` operation.

The location of the user's home directory is shown by the Java `user.home` system property. The location of your Java runtime, shown by the Java system property `java.home`, may vary depending on your platform and installation. Please refer to your JDK or Java runtime installation guide for details.

If you use the `orb.properties` file to specify the ORB class, the layout of this file for AdironORB

would look as follows:

```
org.omg.CORBA.ORBClass=com.adiron.orb.core.ORB
org.omg.CORBA.ORBSingletonClass=com.adiron.orb.core.ORBSingleton
```

The AdironORB installation (or binary distribution) includes a file named `orb.properties` in the `config` directory. This file contains the above two properties, as well as the default settings for AdironORB. You may copy this file to the `lib` subdirectory of your Java runtime or your home directory to use AdironORB.

If you use system properties to specify the ORB, you may pass the required arguments into an application at runtime. This can be done using the `-D` option on the **java** executable. It would look like this:

```
java -Dorg.omg.CORBA.ORBClass=com.adiron.orb.core.ORB \
-Dorg.omg.CORBA.ORBSingletonClass=com.adiron.orb.core.ORBSingleton \
-jar MyApplication.jar
```

If you want to pass properties to the `ORB.init` operation, your application may have code like the following:

```
java.util.Properties properties = new java.util.Properties();

properties.setProperty( "org.omg.CORBA.ORBClass",
    "com.adiron.orb.core.ORB" );
properties.setProperty( "org.omg.CORBA.ORBSingletonClass",
    "com.adiron.orb.core.ORBSingleton" );

org.omg.CORBA.ORB orb =
    org.omg.CORBA.ORB.init( args, properties );
```

At this point, AdironORB is ready to be used. You may further configure the AdironORB using the methods described in Chapter 4.

Using AdironORB as an Avalon Component

When you use AdironORB as an Avalon component, you have to instantiate an ORB instance and invoke the Avalon lifecycle interfaces on your own. In other word, You have to invoke the following methods in sequence on a new ORB instance in order to configure and initialize the ORB properly.

1. `enableLogging`
2. `contextualize`
3. `parameterize`
4. `initialize`

The following code shows an example of creating a server ORB that listens on port 8080.

```
// Create an ORB instance. At this point the ORB is not functional.
com.adiron.orb.core.ORB orb = new com.adiron.orb.core.ORB();

// Create a logger using the Trace class
org.apache.avalon.framework.logger.Logger logger =
    com.adiron.orb.util.Trace.createLogger(
        org.apache.log.Priority.ERROR, "server-orb" );

// Enable the ORB logging
orb.enableLogging( logger );
```

```
// Create a parent context for the ORB.
org.apache.avalon.framework.context.DefaultContext context =
    new org.apache.avalon.framework.context.DefaultContext();
context.makeReadOnly();

// Contextualize the ORB
orb.contextualize( context );

// Create parameters for the ORB
org.apache.avalon.framework.parameters.Parameters params =
    new org.apache.avalon.framework.parameters.Parameters();

// Enable this ORB as a server only
params.setParameter( "com.adiron.orb.client.enable", "false" );
params.setParameter( "iiop.port", "8080" );
params.makeReadOnly();

// Parameterize the ORB
orb.parameterize( params );

// Initialize the ORB
orb.initialize();

// The ORB is functional now.
```

Chapter 4. Configuration

Table of Contents

Overview......15
Ways of Configuring the AdironORB 17
AdironORB Parameters...... 18

Overview

AdironORB provides many ways of configuring the ORB. AdironORB delegates the work of loading, configuring, and initializing the ORB to various objects such as the ORB Loader, ORB Configurator, and ORB Initializer. Each of these objects can be replaced by users' customized object.

AdironORB uses the Avalon parameters to describe the configuration of the ORB. In a CORBA application, these parameters are generated from a list of external sources such as the `orb.properties` files and the arguments and properties supplied to the `ORB.init` operation.

ORB Loader

When a CORBA application or applet uses the `ORB.init` operation to create an ORB instance, AdironORB uses an ORB Loader to initialize the ORB instance Users may, though probably not needed, create their own ORB Loader to customize the ORB initialization process.

Users may pass the `com.adiron.orb.ORBLoaderClass` property to the `ORB.init` operation to override the default ORB Loader. The ORB class takes the value of this property as the class name of the ORB Loader class. An ORB Loader class must implement the `com.adiron.orb.config.ORBLoader` interface. The default value of this property is `com.adiron.orb.config.defimpl.AdironORBLoader`. This default ORB Loader uses the Avalon lifecycle interfaces to configure and initialize the ORB. It also uses an ORB Configurator to derive the configuration of an ORB.

ORB Configurator

When a CORBA application or applet uses the `ORB.init` operation to create an ORB instance, The default ORB Loader, `AdironORBLoader`, uses an ORB Configurator to derive the configuration of the ORB. Users may create their own ORB Configurator to change the way of deriving the ORB configuration. For example, if a user wants to use XML to specify the ORB configuration, the user can replace the default properties-based ORB Configurator with an XML-based ORB Configurator.

Users may pass the `com.adiron.orb.ORBConfiguratorClass` property to the `ORB.init` operation to override the default ORB Configurator. The default ORB Loader takes the value of this property as the class name of the ORB Configurator. An ORB Configurator must implement the `com.adiron.orb.config.ORBConfigurator` interface. The default value of this property is `com.adiron.orb.config.defimpl.PropertiesBasedORBConfigurator`.

The default ORB Configurator, `PropertiesBasedORBConfigurator` derives the configuration of an ORB based on the command line arguments (or applet properties) and the properties supplied to the `ORB.init` operation, as well as some default configurations specified in the `orb.properties` files. We list the sources where the parameters are derived as follows (in the sequence of precedence):

1. The command line arguments (or applet properties) supplied to the ORB.init method: This source has the highest precedence. The default ORB Configurator processes the command line arguments into parameters according to the CORBA Specification, Section 4.5.1. For example, arguments like -ORBxxx yyy will become a parameter named org.omg.CORBA.ORBxxx with value yyy. Arguments of the form -ORBInitRef xxx=yyy are converted into a parameter named org.omg.CORBA.ORBInitRef.xxx with value yyy.

2. The properties supplied to the ORB.init operation: These properties are the second source of parameters. The default ORB Configurator converts properties into parameters in a straightforward way.

3. The system properties whose key begins with com.adiron.orb or org.omg.CORBA: These properties are the third source of parameters. The default ORB Configurator converts these properties into parameters in a straightforward way.

4. The properties defined in the orb.properties file located in the user's home directory or in the lib subdirectory of the Java runtime: If the orb.properties file in the user's home directory is found, the orb.properties file in the lib subdirectory of the Java runtime will be ignored. The location of the Java runtime is specified by the java.home system property.

5. The properties defined in the com/adiron/orb/config/orb.properties file bundeled in the distribution jar file: These properties define the default values of the ORB configuration. A copy of this file is available in the config subdirectory of the AdironORB installation.

Note

Unlike OpenORB, the current implementation of AdironORB does not convert the system properties into the parameters of the ORB configuration. Thus, setting the system properties may not affect the configuration of the ORB unless the system properties are passed to the ORB.init operation. The only exceptions to the above rule are the debugging properties, com.adiron.orb.debug.trace and com.adiron.orb.debug.level, which are checked from the system properties as well.

ORB Initializer

An ORB instance is parameterized when the ORB Loader or the the container component of the ORB passes the ORB configuration to the ORB through the parameterize method. After this point, the ORB will use the parameters of its configuration for the rest life of the ORB, including the initialization of the ORB.

There are two steps in the ORB initialization process: Firstly, the ORB executes an ORB Initializer to initialize the core components of the ORB. Secondly, the ORB executes the Portable Interceptor ORB Initializers, if any, to initialize the standard CORBA plug-ins.

The ORB Initializer to be used by the ORB is determined by the itial-com.adiron.orb.ORBInizerClass parameter. The ORB takes the value of this parameter as the class name of the ORB Initializer class. The default value of this parameter is com.adiron.orb.config.defimpl.SimpleIIOP_ORBInitializer. An ORB Initializer class must implement the com.adiron.orb.config.ORBInitializer interface.

An ORB Initializer is in charge of creating the core components of the ORB. By replacing the ORB Initializer, you can change the effective components of the ORB. For instance, The default ORB Initializer, SimpleIIOP_ORBInitializer, supports only one endpoint (listening port) on a server ORB. You may extend this class to have more endpoints.

Another use of replacing the default ORB Initializer is to support other transport or inter-ORB protocols. For example, you may replace the default ORB Initializer with one that loads a different message engine (instead of GIOP). If you were so inclined, you could have an XML based message engine and run AdironORB as a web service.

Provided in this distribution are the `BaseORBInitializer`, `GIOP_ORBInitializer`, `SimpleIIOP_ORBInitializer`, and `LIOP_ORBInitializer`. The `BaseORBInitializer` sets up a basic ORB that has no capability for inter-ORB communication. The `GIOP_ORBInitializer` adds the support for the GIOP protocol. The `SimpleIIOP_ORBInitializer` plugs in the support for the IIOP protocol. The `LIOP_ORBInitializer`, an experimental ORB Initializer, plugs in a GIOP-over-local-communication protocol that supports inter-ORB communication inside the same process (JVM).

Here is a list of what the `BaseORBInitializer` sets up:

- Policy Factories, Policy Manager, and Policy Current
- DynAny Factory, PI Current, and Request Current
- Client side components: Client Request Manager, Client Channel Manager, and Client Request Interceptor Manager.
- Server side components: Server Manager, IOR Interceptor Manager, Server Request Interceptor Manager, Thread Manager, and RootPOA.
- Delegate Factory, and Binding Factory.
- IOP support: Code Set Database and Codec Factory.

Here is a list of what the `GIOP_ORBInitializer` sets up in addition to what the `BaseORBInitializer` sets up:

- GIOP support: CDR Codec Factory.

Here is a list of what the `SimpleIIOP_ORBInitializer` sets up in addition to what the `GIOP_ORBInitializer` sets up:

- Protocol Connector Factory for IIOP: used to initiate client connections.
- Protocol Acceptor for IIOP: used to receive connections on the server side.

Ways of Configuring the AdironORB

Before we cover what the specific parameters are, and how they affect the operation of the ORB, it is important to understand the different ways to specify parameters to the ORB. The default ORB Configurator, `PropertiesBasedORBConfigurator` allows two ways to specify parameters. The first way is to specify parameters as properties in the `orb.properties` file either in the user's home directory or in the `lib` subdirectory of the Java runtime. The second way is to pass arguments or properties to the `ORB.init` operation.

For example, to set the property `port` in the 'iiop' module to the value '1234' for all orbs on a particular machine, the following line would be added to its `orb.properties` file:

```
iiop.port=1234
```

To ensure that the server is never initialized in a particular application the orb could be initialized as follows:

```
public static void main( String [] args )
{
    java.util.Properties props = new java.util.Properties();
    props.setProperty( "com.adiron.orb.server.enable", "false" );
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB( args, props );
    // ...
}
```

To add a third party portable interceptor, distributed with the ORB Initializer named `org.example.corba.Initializer`, an application can set up properties to the `ORB.init` operation as follows:

```
public static void main( String [] args )
{
    java.util.Properties props = new java.util.Properties();
    props.setProperty( "org.omg.PortableInterceptor.ORBInitializerClass.org.example.Initializer"
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB( args, props );
    // ...
}
```

An alternative way for users to insert a portable interceptor at runtime is to pass the system properties to the `ORB.init` operation as follows:

```
public static void main( String [] args )
{
    org.omg.CORBA.ORB orb =
        org.omg.CORBA.ORB( args, System.getProperties() );
    // ...
}
```

In this way, users wanting to use the above portable interceptor can start the application using the `-D` java argument to define a system property:

```
java -Dorg.omg.PortableInterceptor.ORBInitializerClass.org.example.Initializer \
     app-name
```

AdironORB Parameters

Here is a list of the parameters that the default ORB Initializer and various parts of the ORB understand:

- `com.adiron.orb.debug.trace` - this is the trace level that the system logger will use to determine which kind of messages to output. Acceptable values are: FATAL, ERROR, WARN, INFO, DEBUG.
- `com.adiron.orb.debug.level` - this is the debugging level. This only applies when `com.adiron.orb.debug.trace` is set to DEBUG. The possible values are LOW, MEDIUM, or HIGH.

- `com.adiron.orb.client.enable` - Boolean value which determines whether or not to enable the client side of the ORB.
- `com.adiron.orb.server.enable` - Boolean value which determines whether or not to enable the ORB as a server. If this is false then the ORB may only make client calls to other (server enabled) ORBs.
- `com.adiron.orb.client.bindings.discard_old` - Boolean value which determines whether or not to discard old bindings when the client recieves a `LOCATION_FORWARD`.

  When a client recieves a `LOCATION_FORWARD` reply, the spec demands that the old endpoints should be preserved so that they can be reused in future requests. When this property is set to `true`, old endpoints will be deleted.

  This feature can be used to achieve some kind of load balancing. in this case the IOR always has two endpoints, the first addresses the actual object whereas the second one addresses a load balancing service. Each time the first enpoint fails the second one is used to connect to the load balancing service. The load balancer sends a `LOCATION_FORWARD` reply with two endpoints again. The first one addresses the new object and the second one points to the load balancer again (This is an example on how this feature is currently used).
- `com.adiron.orb.server.shutdownTimeout` - an integer that determines the maximum number of milliseconds to wait before the server starts to cancel the requests that are being serviced. The default value is `1000`.
- `iiop.port` - an integer that tells the port for the server side of the ORB to listen on.
- `iiop.lowPort` - an integer that tells the low end of a range of ports for the server side to listen to. This is used in conjunction with the `iiop.highPort` parameter. If `iiop.port` is set, then this parameter is ignored.
- `iiop.highPort` - an integer that tells the high end of a range of ports for the server side to listen to. This is used in conjunction with the `iiop.lowPort` parameter. If `iiop.port` is set, then this parameter is ignored.
- `com.adiron.orb.server.ThreadManagerClass` - this is the class which acts as the Thread Manager on a server enabled ORB. The default value is `com.adiron.orb.srvmgt.thread.ThreadPoolRequestTM`.
- `com.adiron.orb.server.minThreadSize` - an integer that determines the minimum number of server threads waiting for taking client connections. The default value is `1`.
- `com.adiron.orb.server.maxThreadSize` - an integer that determines the maximum number of server threads allowed. This number will decide the maximum number of concurrent client connections. The default value is `10,000`.
- `com.adiron.orb.server.minDispatchThreadSize` - an integer that determines the minumum number of threads used for dispatching and processing requests. The default value is `1`.
- `com.adiron.orb.server.maxDispatchThreadSize` - an integer that determines the maximum number of threads used for dispatching and processing requests. The default value is `100`.
- `com.adiron.orb.publicContextObjects` - a list of names that denotes a list of context objects (components of an ORB) that are to be accessible through the `licContex-`

`tObject` method. The default value is `BOA,ServerChannelCurrent`.

Note that BOA is not supported by the current AdironORB implementation. However, the name `BOA` is included in this list such that BOA will be supported if a user implements the BOA and registers the BOA as a context object in the ORB Initializer.

Chapter 5. Proprietary API and Policies

Table of Contents

Access the Internals of the ORB 21
Accessing the Transport Information 22
Communicating with Interceptors 24
ORB Policies specific to AdironORB 24

AdironORB provides proprietary extensions to the standard CORBA specification that are needed for certain applications.

Access the Internals of the ORB

AdironORB provides two ways to access some of the internals of the ORB. Firstly, the application code may use the proprietary operations added to the implementation classes of the standard interfaces. Secondly, Portable Interceptor ORB Initializers may use the Avalon `Contextualizable` and `Parameterizable` interfaces to access more of the ORB internals.

The `ORB` Class

The `ORB` class defined in the `com.adiron.orb.core` package provides the following proprietary operations:

- `getParameter`: This method queries the configuration of the ORB as a parameter. It returns the `String` value of the specified parameter.
- `getParameters`: This method returns the configuration of the ORB as an Avalon `Parameters` object. Users may query the returned `Parameters` object using its interface.
- `getPublicContextObject`: This method retrieves an internal object that has been made public by the ORB, e.g., `ServerChannelCurrent`. This method is provided because these internal objects do not implmement the `org.omg.CORBA.Object` interface.
- `addServiceInformation`: This method adds the service information to the ORB such that this information will be available through the `get_service_information` operation.

The `ORBInitInfo` Object

When a Portable Interceptor (PI) ORB Initializers' `pre_init` or `post_init` operation is invoked, an object implementing the `PortableInterceptor::ORBInitInfo` interface is passed to provide service and information of the ORB to the ORB Initializer. AdironORB adds the proprietary `orb` operation to this object such that an ORB Initializer may retrieve the reference to the ORB instance. Users may type-cast this object to the `com.adiron.orb.core.ORBInitInfo` interface and invoke the `orb` operation.

Theoretically, an ORB instance is not a fully functional ORB during the execution of PI ORB Initializers. In reality, however, there are many reasons for PI ORB Initializers to refer to the ORB instance at this early stage of an ORB.

The Avalon Extension to ORB Initializers

AdironORB provides PI ORB Initializers with access to the internals of an ORB using the Avalon framework. For example, if an ORB Initializer implements the Avalon's `Contextualizable` interface, a `Context` object that contains references to many ORB internal objects will be passed through the `contextualize` method. Similarly, the configuration of an ORB will be passed through the `parameterize` method if the `Parameterizable` interface is implemented.

The following code shows an example ORB Initializer that implements the `Contextualizable` interface. The ORB Initializer can detect whether the server-side of the ORB is enabled or not.

```
public class MyORBInitializer
    extends    org.omg.CORBA.LocalObject
    implements org.omg.PortableInterceptor.ORBInitializer,
               org.apache.avalon.framework.context.Contextualizable
{
    public void contextualize(
        org.apache.avalon.framework.context.Context context
    )
        throws org.apache.avalon.framework.context.ContextException
    {
        // The ORB reference is always available
        com.adiron.orb.core.ORB orb =
            ( com.adiron.orb.core.ORB )
                context.get( "ORB" );

        // Try to get the Server Manager of the ORB
        com.adiron.orb.server.ServerManager serverManager = null;
        try
        {
            serverManager =
                ( com.adiron.orb.server.ServerManager )
                    context.get( "ServerManager" );
        }
        catch( org.apache.avalon.framework.context.ContextException ex )
        {
            // fall through
        }

        if ( serverManager == null )
        {
            // The server-side of the ORB is disabled.
        }
    }
    // ...
}
```

Accessing the Transport Information

The CORBA framework provides a location-transparent way for applications, but it is sometimes necessary to have the knowledge about the underlying network and transport that are used to carry out object invocations. For example, in the needs of security, it is desireable to have the underlying knowledge of the transport such as the IP, port number, authenticated X.509 certificate chain, etc. AdironORB provides two way to retrieve this information: one is for the client side and the other is for the server side.

Accessing the Client-Side Transport Information

A client using the AdironORB can access the information about the underlying transport that is used by an object reference to connect to a target object. The following code gives an example of retrieving this

information.

```
org.omg.CORBA.portable.ObjectImpl objImpl =
    ( org.omg.CORBA.portable.ObjectImpl ) obj;

com.adiron.orb.request.Delegate objDelegate =
    ( com.adiron.orb.request.Delegate )
        objImpl._get_delegate();

com.adiron.orb.request.ObjRefBinding binding =
    objDelegate.getSelectedBinding();

// If the binding is null, this object reference is probably
// not connected to the target.
if ( binding == null )
{
    // Try to make an invocation.
    obj._non_existent();
    binding = objDelegate.getSelectedBinding();
    if ( binding == null )
    {
        throw new RuntimeException( "Delegate does not have a binding" );
    }
}

com.adiron.orb.client.ClientChannelInfo channelInfo =
    binding.getClientChannelInfo();

// If channelInfo is null, it is a local binding.
if ( channelInfo != null )
{
    com.adiron.orb.transport.TransportInfo transportInfo =
        channelInfo.getTransportInfo();

    // If transportInfo is null, it is closed or not connected.
    if ( transportInfo != null )
    {
        System.out.println( transportInfo.describe() );
    }
}
```

Accessing the Server-Side Transport Information

AdironORB provides the object implementations to retrieve information about the underlying transport on which a request is received using the Server Channel Current object. A Server Channel Current object, implementing the `com.adiron.orb.server.ServerChannelCurrent` interface, provides information about the underlying transport on which a server channel is created.

Each ORB instance has its own Server Channel Current object, which can be retrieved through the `ORB.getPublicContextObject` operation. This Server Channel Current object may also be retrieved from the `Context` object passed to an PI ORB Initialize through the `Contextualizable` interface.

The following code shows an example of retrieving the transport information from an object implementation.

```
com.adiron.orb.core.ORB apacheORB = ( com.adiron.orb.core.ORB ) orb;

com.adiron.orb.server.ServerChannelCurrent serverChannelCurrent =
    ( com.adiron.orb.server.ServerChannelCurrent )
        apacheORB.getPublicContextObject( "ServerChannelCurrent" );

com.adiron.orb.server.ServerChannelInfo channelInfo =
```

```
        serverChannelCurrent.getServerChannelInfo();

    // If channelInfo is null, it is a local call.
    if (channelInfo != null)
    {
        com.adiron.orb.transport.TransportInfo transportInfo =
            channelInfo.getTransportInfo();
        System.out.println( transportInfo.describe() );
    }
```

Communicating with Interceptors

The CORBA specification does not provide a standard way for application code to communicate with Portable Interceptors. AdironORB provides an extension to its Delegate implementation to associate information objects with a Delegate. These information objects are added to a Delegate in the per-thread bases. That is, objects added by a thread can be retrieved or removed only by the same thread.

The `com.adiron.orb.request.Delegate` class defines three methods for this purpose:

- `addPerThreadInfo`: This method adds a per-thread information object to a Delegate.
- `getPerThreadInfo`: This method retrieves a per-thread information object from a Delegate.
- `removePerThreadInfo`: This method retrieves and removes a per-thread information object from a Delegate.

ORB Policies specific to AdironORB

There are two policies specific to the AdironORB that will affect the runtime behavior of the ORB.

- `ForceMarshal`: This policy takes a boolean value as an argument. If the argument is true, then local calls are not allowed. This means that every call will be marshalled by the ORB. This is used to force the interception of ALL calls made on this ORB. The default value is false for the ForceMarshal policy.
- `InterceptLocal`: This policy takes a boolean value as an argument. If the argument is true, then local calls will be intercepted by the ORB. This is accomplished with a dynamic proxy object that wraps the implementation object. Currently, only synchronous calls are supported using this dynamix proxy.

The interaction between the two policies is quite simple. The ForceMarshal policy will always override the InterceptLocal policy.

Chapter 6. References

- Adiron LLC is a foundation exists to provide organizational, legal, and financial support for the Apache open-source software projects. Adiron LLC [http://www.adiron.com].
- Ant is the Apache project's Java build tool. There is extensive documentation available on the Ant website [http://jakarta.apache.org/ant/index.html].
- The Apache Avalon Framework [http://jakarta.apache.org/avalon] interfaces and design methodology is used throughout the internal code of AdironORB.
- The Java extension mechanism is used to add jar files to a Java installation as if they were a core part of the product. Sun has online documentation [http://java.sun.com/docs/books/tutorial/ext/] available.
- The method of specifying Portable Interceptor ORB Initializers as keys is described in Sun's Javadoc [http://java.sun.com/j2se/1.4/docs/api/org/omg/PortableInterceptor/ORBInitializer.html].
- The Community OpenORB was the origin of what has become AdironORB. Information can be found on the Community OpenORB website [http://openorb.sf.net].

Appendix A. AdironORB Error Codes

The following list provides the explanation of AdironORB minor error codes for CORBA System exceptions. For information concerning OMG Standard Minor Codes, see the following URL: http://www.omg.org/cgi-bin/doc?minor-codes.

- VMCID = 0x414f0000 (1095696384)
- BAD_INV_BASE_VALUE = 10
- 11 : Attempt to access incomplete typecode containing recursive
- 12 : Invocation order when using DSI
- 13 : Invocation order when using streaming stubs (Delegate)
- 14 : Orb is not initialized
- 15 : Server invocation order
- 16 : The current reply status is not valid at this intercept point.
- BAD_OPERATION_BASE_VALUE = 20
- 21 : Attempt to extract wrong type from any
- 22 : CDR version does not support primitive
- BAD_PARAM_BASE_VALUE = 30
- 31 : Object class cannot be instantiated or is incorrect type
- 32 : Type mismatch in list streams with fixed type
- 33 : Object class cannot be instantiated or is incorrect type
- 34 : Attempt to insert value into any with incorrect typecode
- 35 : Array index out of bounds
- 36 : Object class cannot be instantiated or is incorrect type
- 37 : No primitive typecode of that kind
- 38 : Null valued strings cannot be transmitted
- COMM_FAILURE_BASE_VALUE = 50
- 51 : Connection to client has been lost before reply can be sent
- 52 : No route to server
- 53 : No connection to server, server is not listening or connection refused
- 54 : Unable to find host in DNS

Appendix A. AdironORB Error Codes

- 55 : IOException occoured during read
- 56 : Unexpected end of stream during read
- 57 : Broken data during read
- 58 : Message error. Remote server detected a broken AdironORB
- INF_REPOS_BASE_VALUE = 70
- 71 : Unable to find interface repository
- 72 : Unable to find interface in interface repository
- 73 : Unable interface from repository is the wrong type
- INV_OBJREF_BASE_VALUE = 80
- 81 : Invalid tag for profile
- 82 : Profile data is corrupted
- 83 : Component data is corrupted
- 84 : Component data is corrupted
- INV_POLICY_BASE_VALUE = 90
- 91 : Object class cannot be instantiated or is incorrect type
- MARSHAL_BASE_VALUE = 100
- 101 : Problem with marshaling or unmarshalling char data
- 102 : Problem with marshaling or unmarshalling wchar data
- 103 : Recursive typecode offset does not match any known typecode
- 104 : Typecode kind unknown
- 105 : Problem with fixed type
- 106 : Problem with valuetype encoding
- 107 : Failed to close encapsulation layer before calling close operation
- 108 : Sequence length exceeds limit in typecode
- 109 : Type mismatch for list stream
- 110 : Bounds mismatch for list stream
- 111 : Buffer overread
- 112 : Buffer underread

- 113 : Invalid buffer position or format.
- 114 : Attempt to insert native type into any
- 115 : Problem with union discriminator
- 116 : Unable to extract valuebox type from any, missing helper
- 117 : Unreported exception occurred during marshalling a request.
- 118 : Unreported exception occurred during marshalling or the reply buffer is underread.
- NO_RESOURCES_BASE_VALUE = 130
- 131 : Problem with valuetype encoding
- NO_PERMISSION_BASE_VALUE = 140
- 141 : The port published in the IOR was 0. This is probably a bidir only target
- UNKNOWN_BASE_VALUE = 150
- 151 : An interceptor raises a non-compliant exception.
- 152 : An interceptor throws ForwardRequest on a co-located call (local invocation).

The following list provides the OMG error codes for the CORBA System Exceptions.

UNKNOWN

- 1 : Unlisted user exception received by client
- 2 : Non-standard System Exception not supported
- 3 : An unknown user exception received by a portable interceptor

BAD_PARAM

- 1 : Failure to register, unregister, or lookup value factory
- 2 : RID already defined in IFR
- 3 : Name already used in the context in IFR
- 4 : Target is not a valid container
- 5 : Name clash in inherited context
- 6 : Incorrect type for abstract interface
- 7 : string_to_object conversion failed due to bad scheme name
- 8 : string_to_object conversion failed due to bad address

- 9 : string_to_object conversion failed due to bad schema specific part
- 10 : string_to_object conversion failed due to non-specific reason
- 11 : Attempt to derive abstract interface from non-abstract base interface in the Interface Repository
- 12 : Non-abstract interface in the Interface Repository
- 13 : Attempt to use an incomplete TypeCode as a parameter
- 14 : Invalid object id passed to POA::create_reference_by_id
- 15 : Bad name argument in TypeCode operation
- 16 : Bad RepositoryId argument in TypeCode operation
- 17 : Invalid member name in TypeCode operation
- 18 : Duplicate label value in create_union_tc
- 19 : Incompatible TypeCode of label and discriminator in create_union_tc
- 20 : Supplied discriminator type illegitimate in create_union_tc
- 21 : Any passed to ServerRequest::set_exception does not contain an exception
- 22 : Unlisted user exception passed to ServerRequest::set_exception
- 23 : wchar transmission code set not in service context
- 24 : Service context is not in OMG-defined range
- 25 : Enum value out of range

IMP_LIMIT

- 1 : Unable to use Any profile in IOR

INV_OBJREF

- 1 : wchar Code Set support not specified

MARSHAL

- 1 : Unable to locate value factory
- 2 : ServerRequest::set_result called before ServerRequest::ctx when the operation IDL contains a context clause
- 3 : NVList passed to ServerRequest::arguments does not describe all parameters passed by client
- 4 : Attempt to marshal Local object

Appendix A. AdironORB Error Codes

BAD_TYPECODE

- 1 : Attempt to marshal incomplete TypeCode
- 2 : Member type code illegitimate in TypeCode operation

NO_IMPLEMENT

- 1 : Missing local value implementation
- 2 : Incompatible value implementation version
- 3 : Unable to use any profile in IOR
- 4 : Attempt to use DII on Local object

BAD_INV_ORDER

- 1 : Dependency exists in IFR preventing destruction of this object
- 2 : Attempt to destroy indestructible objects in IFR
- 3 : Operation would deadlock
- 4 : ORB has shutdown
- 5 : Attempt to invoke send or invoke operation of the same Request object more than once
- 6 : Attempt to set a servant manager after one has already been set
- 7 : ServerRequest::arguments called more than once or after a call to ServerRequest:: set_exception
- 8 : ServerRequest::ctx called more than once or before ServerRequest::arguments or after rRe-Servequest::ctx, ServerRequest::set_result or ServerRequest::set_exception
- 9 : ServerRequest::set_result called more than once or before ServerRequest::arguments or after ServerRequest::set_result or ServerRequest::set_exception
- 10 : Attempt to send a DII request after it was sent previously
- 11 : Attempt to poll a DII request or to retrieve its result before the request was sent
- 12 : Attempt to poll a DII request or to retrieve its result after the result was retrieved previously
- 13 : Attempt to poll a synchronous DII request or to retrieve results from a synchronous DII request

TRANSIENT

- 1 : Request discarded due to resource exhaustion in POA
- 2 : No usable profile in IOR

Appendix A. AdironORB Error Codes

OBJ_ADAPTER

- 1 : System exception in POA::unknown_adapter
- 2 : Servant not found [ServantManager]
- 3 : No default servant available [POA policy]
- 4 : No servant manager available [POA Policy]
- 5 : Violation of POA policy by ServantActivator::incarnate

DATA_CONVERSION

- 1 : Character does not map to negotiated transmission code set

OBJECT_NOT_EXIST

- 1 : Attempt to pass an unactivated (unregistered) value as an object reference
- 2 : POAManager::incarnate failed to create POA