Java Interoperability Guide

iPlanet™ Unified Development Server

Version 5.0

August 2001

Contents

List of Figures

List of Procedures

Preface

The *iPlanet UDS Java Interoperability Guide* explains how you use Common Object Request Broker Architecture (CORBA) to write objects that interact transparently with other remote objects that are written in different languages and run on different operating systems and hardware.

This preface contains the following sections:

Product Name Change

Forte 4GL has been renamed the iPlanet Unified Development Server. You will see full references to this name, as well as the abbreviations iPlanet UDS and UDS.

Audience for This Guide

This manual is intended for application developers. We assume that you:

- have TOOL programming experience
- are familiar with your particular window system
- understand the basic concepts of object-oriented programming as described in *A Guide to the iPlanet UDS Workshops*
- have used the iPlanet UDS workshops to create classes

Organization of This Guide

The following table briefly describes the contents of each chapter:

Chapter	Description
Chapter 1, "Interoperating With CORBA Objects"	Provides an overview of the CORBA architecture and the TOOL libraries and utilities that you use to support it.
Chapter 2, "CORBA Servers"	Provides an overview of CORBA servers and then examines a Java and TOOL server implementation as examples.
Chapter 3, "Working with CORBA Clients"	Provides an overview of CORBA clients and then examines a Java and TOOL client implementation as examples.
Appendix A, "IDL and TOOL"	Describes the mapping between IDL types and TOOL types, and between TOOL types and IDL types.
Appendix B, "Using Fscript to Configure CORBA Servers"	Gives detailed information for script writers about the settings you can use with service objects that are acting as CORBA servers.

Text Conventions

This section provides information about the conventions used in this document.

Format	Description
italics	Italicized text is used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.
ALL CAPS	Text in all capitals represents environment variables (FORTE_ROOT) or acronyms (UDS, JSP, iMQ).
	Uppercase text can also represent a constant. Type uppercase text exactly as shown.
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

Other Documentation Resources

In addition to this guide, there are additional documentation resources, which are listed in the following sections. The documentation for all iPlanet UDS products (including Express, WebEnterprise, and WebEnterprise Designer) can be found on the iPlanet UDS Documentation CD. Be sure to read "Viewing and Searching PDF Files" on page 16 to learn how to view and search the documentation on the iPlanet UDS Documentation CD.

iPlanet UDS documentation can also be found online at http://docs.iplanet.com/docs/manuals/uds.html.

The titles of the iPlanet UDS documentation are listed in the following sections.

iPlanet UDS Documentation

- *A Guide to the iPlanet UDS Workshops*
- *Accessing Databases*
- *Building International Applications*
- *Escript and System Agent Reference Guide*
- *Fscript Reference Guide*
- *Getting Started With iPlanet UDS*
- *Integrating with External Systems*
- *iPlanet UDS Java Interoperability Guide*
- *iPlanet UDS Programming Guide*
- *iPlanet UDS System Installation Guide*
- *iPlanet UDS System Management Guide*
- *Programming with System Agents*
- *TOOL Reference Guide*
- *Using iPlanet UDS for OS/390*

Express Documentation

- *A Guide to Express*
- *Customizing Express Applications*
- *Express Installation Guide*

WebEnterprise and WebEnterprise Designer Documentation

- *A Guide to WebEnterprise*
- *Customizing WebEnterprise Designer Applications*
- *Getting Started with WebEnterprise Designer*
- *WebEnterprise Installation Guide*

Online Help

When you are using an iPlanet UDS development application, press the F1 key or use the Help menu to display online help. The help files are also available at the following location in your iPlanet UDS distribution: `FORTE_ROOT/userapp/forte/cln/*.hlp`.

When you are using a script utility, such as Fscript or Escript, type help from the script shell for a description of all commands, or help `<command>` for help on a specific command.

iPlanet UDS Example Programs

A set of example programs is shipped with the iPlanet UDS product. The examples are located in subdirectories under `$FORTE_ROOT/install/examples`. The files containing the examples have a `.pex` suffix. You can search for TOOL commands or anything of special interest with operating system commands. The `.pex` files are text files, so it is safe to edit them, though you should only change private copies of the files.

Viewing and Searching PDF Files

You can view and search iPlanet UDS documentation PDF files directly from the documentation CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

NOTE You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from http://www.adobe.com. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

➤ To copy the documentation to a client or server

1. Copy the `doc` directory and its contents from the CD-ROM to the client or server hard disk.

   You can specify any convenient location for the `doc` directory; the location is not dependent on the iPlanet UDS distribution.

2. Set up a directory structure that keeps the `udsdoc.pdf` and the `uds` directory in the same relative location.

   The directory structure must be preserved to use the Acrobat search feature.

NOTE To uninstall the documentation, delete the `doc` directory.

➤ To view and search the documentation

1. Open the file `udsdoc.pdf`, located in the `doc` directory.
2. Click the Search button at the bottom of the page or select Edit > Search > Query.

3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click Search.

   A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

   NOTE For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

   All occurrences of the word or phrase on a page are highlighted.

5. Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

Toolbar Button	Keyboard Command
Next Highlight	Ctrl+]
Previous Highlight	Ctrl+[
Next Document	Ctrl+Shift+]

   To return to the `udsdoc.pdf` file, click the Homepage bookmark at the top of the bookmarks list.

6. To revisit the query results, click the Results button at the bottom of the `udsdoc.pdf` home page or select Edit > Search > Results.

Chapter 1

Interoperating With CORBA Objects

Using Common Object Request Broker Architecture (CORBA) you can write objects that interact with remote objects without your needing to be concerned about where the objects are located, what language they are written in, or how they perform their tasks.

Using the iPlanet UDS Workshops, you can include libraries that provide CORBA support and you can create projects that contain classes whose methods request operations on external objects. These classes are stored in the repository and used in TOOL just like any other class.

This chapter provides a brief overview of the CORBA architecture and the TOOL libraries and utilities that you use to support this architecture.

Overview

CORBA is a standard distributed object architecture that allows application components written by different vendors to interoperate across networks and operating systems. Objects that interoperate in this way are called CORBA objects. These objects can be written in different languages; they communicate by using the Interface Definition Language (IDL), which defines their interfaces.

CORBA Architecture and Concepts

The CORBA architecture and concepts introduced in this section are defined in the *CORBA/IIOP 2.0 Specification*. This section provides a brief overview of this material.

Basic Concepts

An application using CORBA's architecture to allow objects, written in different languages and hosted on different operating systems and machines, to interoperate consists of the following components:

- A *client:* the code where a method invocation originates. A client uses an ORB to access services from the ORB of the server. (Clients can also be referred to as CORBA clients or IIOP clients.)
- An *object implementation:* the code that receives the method invocation; it is part of a local or remote server. Each object implementation has a unique *object reference* that the client uses to find the object and invoke its methods. (An object implementation can also be referred to as a *CORBA object*.)

  The object implementation or CORBA object is part of a server that uses an ORB to provide services to a client.
- An IDL (Interface Definition Language) file: a text file containing IDL statements that specify the interface to an object implementation. The IDL interface definition is independent of any single programming language, but maps to all of the popular programming languages, using standardized mappings for Java, C++, C, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript.

  The IDL interface defines the object's type, specifies the signatures of the methods that the object implements, and specifies the exceptions that the methods return. Both the client and the server must compile this text file, which generates an IDL stub file, an IDL skeleton file, and other files required for CORBA support.

  - An IDL stub file, or *stub*, is used by the client. It translates the object implementation's interface from a language-neutral format to the client's implementation language. The stub files contain generated code that handles the marshalling of method invocations into a form that can be transmitted over the wire.
  - An IDL skeleton file, or *skeleton*, is used by the server. It maps the language-neutral form of the CORBA object's interface to the language in which the CORBA object is implemented. The skeleton files contain generated code that turns requests coming in over the wire into the format expected by the server.

    Depending on how you develop the server, you can either write your own IDL file or have the iPlanet UDS runtime create one for you at partitioning time. For more information, see "Developing and Building Servers" on page 43.

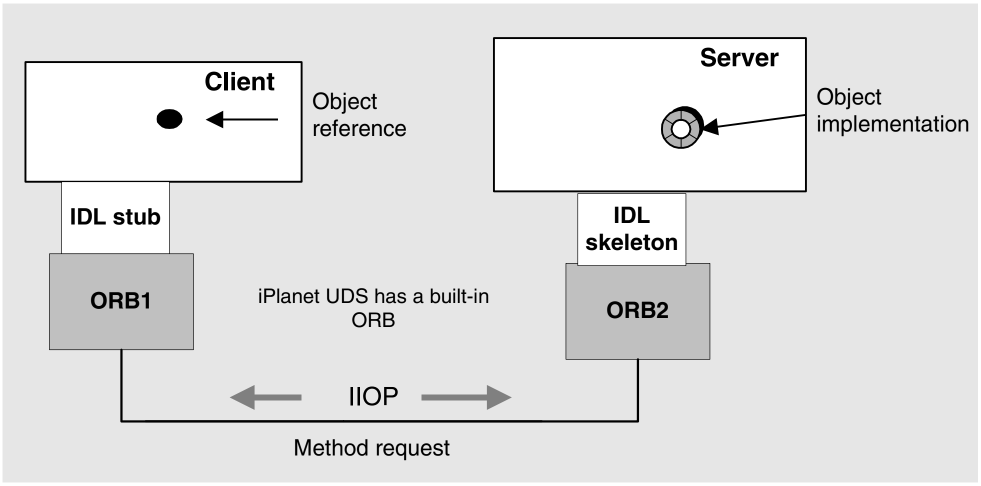
For more information about the OMG Interface Definition Language (IDL), see *Interface Description Language: Definition and Use* by Richard Snodgrass, Computer Science Press, 1989 or Chapter 3 of the Common Object Request Broker: Architecture and Specification Revision 2.2.

- An ORB (Object Request Broker): the communication infrastructure supplied by the vendor for a particular run-time system. ORBs use CORBA 2.0 and IIOP to serve as middle ware for the client and server. They find the object implementation, they package a method call as a message, they interpret the message as a method call (for the server), and they enable the server to invoke the correct method on the correct object and pass any return values back to the client. iPlanet UDS has a built-in ORB.
- An IOR (Interoperable Object Reference) file: a file provided by the server. The file contains an IOR string that can be interpreted by the client's ORB to locate and bind to a particular object implementation. If you use the naming service to get an object reference, you do not need to create or use this file. For more information, see "Getting an Object Reference" on page 22.

Architecture

Figure 1-1 shows how the entities described in the previous section work together to allow distributed objects to interoperate.

Figure 1-1 CORBA Architecture



Having obtained an object reference for the object, the client uses the reference to invoke a method on the object. The client uses the same code it would use if this were a local method call, but substitutes the object reference for the class instance name. If the object implementation is remote, the object reference points to a stub, which uses the ORB to forward the invocation to the object. The stub uses the ORB to identify the machine where the object is running and asks that machine's ORB for a connection to the object's server. The stub then sends the object reference and actual parameters to the skeleton code that is linked to the remote object. The skeleton transforms the call and parameters into the format expected by the CORBA object, and calls the method. Any results or exceptions are returned along the same path.

Although you do need to generate the stub and skeleton from the IDL file that describes the object's interface, and to obtain the object reference, you do not have to concern yourself with how the stubs, skeleton, and ORBs work together to enable transparent communication.

Getting an Object Reference

Before a client can call a CORBA object, it must obtain a reference to the object. An *object reference* is an opaque structure that identifies the CORBA object's host machine and the port on which the host server is listening for requests. The object reference also contains a key that identifies the actual object on the server. Having obtained the object reference, the client can then invoke a method on the object by using that reference.

The client can get an object reference in one of two ways: using the naming service or using an IOR file.

- The client can start a COS-compliant (Common Object Services) naming service and call an ORB method that resolves a name into an object reference. (In this case, the server has already registered the object implementation with the naming service.) For more information see "Using the Naming Service" on page 64.
- The server uses an ORB method to create a string from an object reference; the server can then write the object reference to an IOR file and the client can read the file and call another ORB method to convert the string to an object reference. For more information, see "Using an IOR File to Pass an Object Reference" on page 47.

- At partitioning time, the server developer can mark the service object as a CORBA object and then use the ensuing IIOP Configuration dialog to specify the name of an IOR file. If this option is used, the iPlanet UDS runtime creates an IOR file for the client's use either at runtime or when the application distribution is made. The client can then use an ORB method to turn the IOR string into an object reference.

  This is similar to the second method except for the fact that the IOR file gets generated automatically and offers the option to configure a server in a suitable way if there is a firewall. For more information, see "Using the IIOP Configuration Dialog" on page 48.

Using the first method is recommended for intranet interoperability, using the last two methods is preferable for situations in which multiple external clients are accessing one or more servers across a firewall.

Naming Service

If the ORB vendor provides a CORBA COS Naming Service, it will be organized as a tree-like directory or *namespace* for object references. The namespace is organized like a file system that provides directories in which to store files. The root of the namespace is called the *initial naming context;* subdirectories in the name space are called *naming contexts*. When you store an object reference in a naming context, you associate it with an advertise name and the object reference-name pair is called a name *binding*.

ORB vendors often provide an implementation of the CORBA COS Naming Service. A server can use the naming service to register the CORBA objects it wants to advertise; the client can use the naming service to obtain a reference for the CORBA objects whose methods it wants to call. You must explicitly start the naming service and assign it a port number before clients or servers can use it.

For additional information on how to use the naming service to obtain an object reference, see the code examples shown in Chapter 2 and Chapter 3 of this manual. For code samples that relate to specific ORB providers, see Technical Note 12500.

CORBA Support

TOOL support for CORBA is, for the most part, integrated into the iPlanet UDS runtime system. In addition to the system support, iPlanet UDS also supplies the following:

- The means to generate an IDL-to-TOOL mapping from an IDL text file.
- A runtime IIOP transport that takes the place of the stub and the skeleton.

- The CosNaming library, a library that supplies the classes you need to access and use a naming service.
- The CORBA library, which provides the ORB class, the TOOL interface CorbaObject (which maps to the IDL type object), and the CorbaObjectImpl class, which is the root class for all CORBA classes.

  Note that the UDS runtime does not require an ORB to communicate.

You can use the iPlanet UDS CORBA support to create a TOOL server that can interoperate with non-TOOL clients or to create a TOOL client that can interoperate with non-TOOL servers. The examples given in this book focus on TOOL-Java interoperability, but you can use the TOOL CORBA library and the built in runtime support to communicate with any other ORB-enabled client or server.

CORBA and CosNaming interface specifications are currently published at the following locations:

- ftp://ftp.omg.org/pub/docs/formal/97-12-10.pdf
- ftp://ftp.omg.org/pub/docs/formal/97-12-10.ps

The TOOL CORBA Library

The table below describes the contents of the TOOL CORBA library.

Name	**Superclass**	**Description**
CorbaObject	Interface	Provides the mapping to the IDL type Object. Any object that can be passed as a parameter of type Object or that can be received as a return type or output parameter whose declared type is Object needs to implement the CorbaObject interface. This applies to any object using the naming service. Any classes that inherit from CorbaObjectImpl automatically implement this interface.

Name	Superclass	Description
CorbaObjectImpl	Object	Implements CorbaObject. Recommended as base class for all TOOL classes that are used with CORBA. CorbaObjectImpl is returned for a return value or output parameter when its declared IDL type is Object and if the actual type returned is not known in the current project. When the return type is Object in IDL and CorbaObject in TOOL users should always use ORB.narrow to make sure they get an object of the desired type. CorbaObjectImpl is mapped to Framework::Object in IDL.
ORB	Object	A class providing APIs for the CORBA Object Request Broker features. --Initializes the ORB implementation by supplying values for predefined properties and environmental parameters --Obtains initial object references to services such as the Naming Service using the method resolve_initial_references --Converts object references to strings and back
BAD_OPERATION	SystemException	Exception class that is raised by certain ORB operations.
BAD_PARAM	SystemException	Exception class that is raised by certain ORB operations.
InvalidName	UserException	Exception class that is raised by certain ORB operations.

The CosNaming Library

The table below describes the contents of the TOOL CosNaming library.

Name	Superclass	Description
Binding	Object	A class that represents the association of a name with an object. The naming context is populated by one or more bindings.

Name	Superclass	Description
BindingIterator	CorbaObjectImpl	A class whose methods allow you to iterate through all bindings in the naming service.
NameComponent	Object	A class that provides the structure of the name hierarchy.
NamingContext	CorbaObjectImpl	A class that you use to associate a name with an object reference, to obtain an object reference from a given name, and to create or destroy nodes in the naming service hierarchy.
AlreadyBound	UserException	Exception class that is raised when you are trying to register a name that has already been registered.
CannotProceed	UserException	Exception class, general purpose.
InvalidName	UserException	Exception class that is raised by passing a 0-length name or a name that otherwise violates a CosNaming implementation; for example, a name containing spaces, etc.
NotEmpty	UserException	Exception class that is raised when you attempt to destroy a non-empty naming context.
NotFound	UserException	Exception class that is raised when a name is looked up in the naming service that does not exist.

Using ORB Development Tools

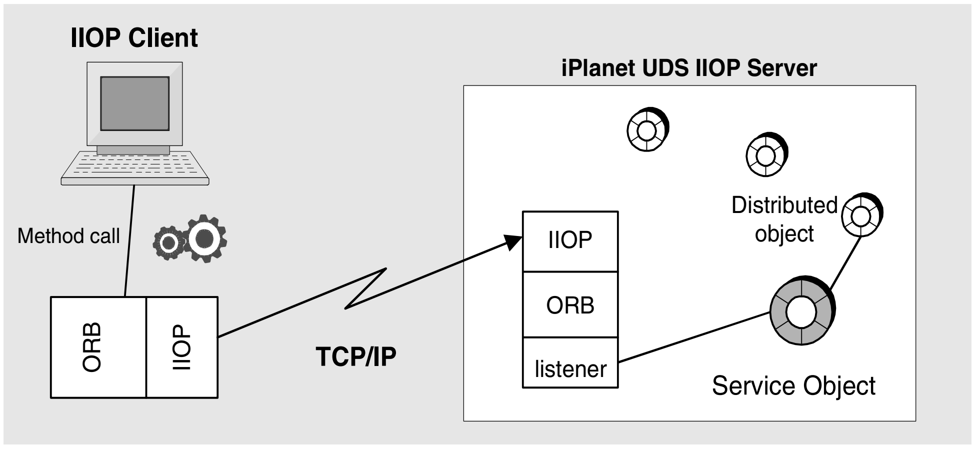
Typically, you use an ORB development tool to develop the non-TOOL client or server that communicates with a TOOL client or server. Although iPlanet UDS supports ORB products supplied by both Visigenic and Iona, this manual does not provide product-specific information about using them. Please see Technical Note 12500 for an index to other technical notes that contain code samples and information about working with these vendors.

You can access the iPlanet UDS Technical Notes either through the iPlanet UDS web site http://www.forte.com/support/technotes.html) Tech Info pages, or by calling iPlanet UDS Technical Support.

TOOL Server Design

A TOOL *server* is an application with at least one object that has been made available to one or more CORBA clients. The section "Getting an Object Reference" on page 22 explains how an object makes itself available to a client. Typically the object is a service object, but it does not have to be. The implementation of TOOL servers is examined in detail in the chapter "CORBA Servers" on page 41. The following subsections focus on how messages are conveyed to the TOOL server through a *listener* and describe your design options when configuring a listener.

Figure 1-2 iPlanet UDS IIOP Server



As you can see in Figure 1-2, there is another service that is involved in communicating with a TOOL/CORBA server, the *listener*. The listener is connected to a particular TCP/IP address and port and uses the CORBA infrastructure to accept and respond to client messages. It communicates with objects inside the iPlanet UDS environment by using standard iPlanet UDS protocols.

Accessing other distributed objects The listener contacts the service object first. If the client application calls a service object method and obtains a reference to another distributed object, subsequent calls to the distributed object also go through the listener, but the listener can then communicate directly with the distributed object. (The client can call the distributed object directly.)

This is not the case for any objects that have been advertised in the naming service or for which object_to_string has been called before they were passed out by the service object. A call to these objects would go directly to the object through a listener local to those objects, not through the service object's listener. Thus using the naming service or object_to_string should be avoided if calls to these objects need to go through the same listener as the service object

Accessing multiple service objects Figure 1-2 shows only one service object. However, you could have multiple service objects in an iPlanet UDS environment, all of which are known to the client and all of which are directly accessible through IIOP. You can associate each CORBA service object with its own listener, or associate a single listener with multiple service objects. There is a limit of one listener per partition, so if you choose to have one listener for each service object, each service object must be in its own partition. (To associate each service object with its own listener, you must use the IIOP configuration dialog as shown in "One Listener Per Service Object, Same Partition" on page 29.)

If you are using an IOR string to access a service object that has been configured using the IIOP configuration dialog, each object must be listed in a separate IOR file to be directly accessible. (The IOR string provided in the IOR file contains both the TCP/IP host and port for a listener and specifies which service object the listener is to contact.)

Stand-alone listeners If, for security reasons, you want to set up a single listener for all your CORBA service objects, you can use the IIOP Gateway application. This application is provided for the following circumstances:

- It can provide a single point at which an iPlanet UDS environment receives IIOP requests, thus satisfying the requirements of Java security.
- If your TOOL server requires a listener on a separate partition, iPlanet UDS starts an IIOP gateway on that partition, and the gateway starts the listener.

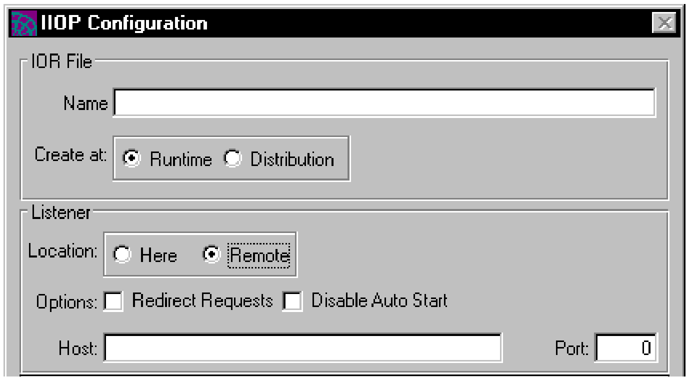
For more information about the IIOP gateway, see "Using the iPlanet UDS IIOP Gateway" on page 34.

Listener and Service Object Configurations

When a client uses an object reference to invoke a method on a TOOL server, the listener routes the request to the appropriate service object. You can use a single listener to access multiple service objects, or you can use multiple listeners. A listener can be on the same partition as the service object it accesses, or it can be on a remote partition. In order to control the configuration of service objects and listeners, you must use the IIOP configuration dialog for the service object. This is especially important if you need to set up a server for load balancing or failover.

The configuration dialog is shown in Figure 1-3. The sections that follow describe a number of listener-service object configurations. Each section provides brief information about the configuration parameters for the service objects. You would use the configuration dialog to set these parameters.

Figure 1-3 IIOP Configuration Dialog


IIOP Configuration

IOR File

Name:

Create at: [✓ Runtime] Distribution

Listener

Location: Here [✓ Remote]

Options: Redirect Requests Disable Auto Start

Host:

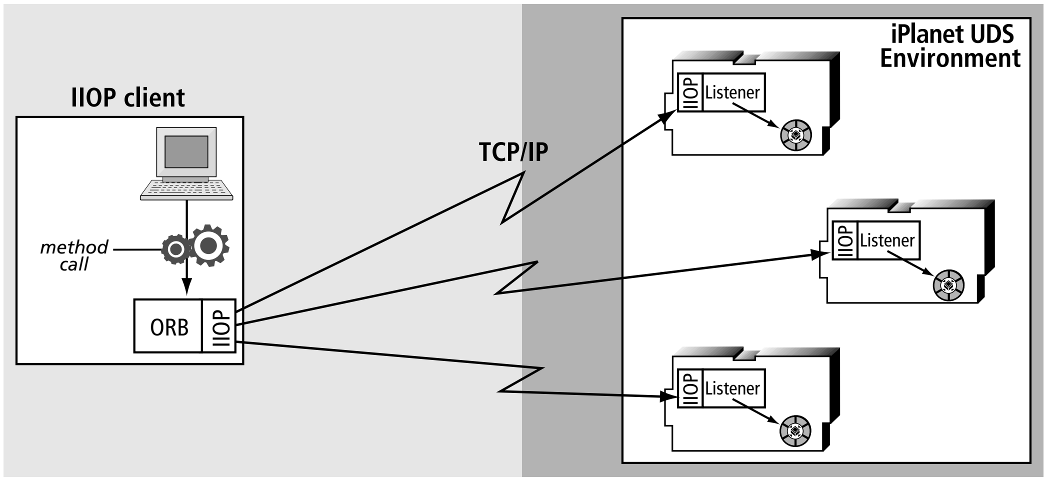
Port: 0

For complete information about configuration parameters and how to set them in the Partition Workshop, see "Using the IIOP Configuration Dialog" on page 48. For information about setting these configuration parameters using the Fscript `SetServiceEOSInfo` command, see "Configuration Parameters for a TOOL CORBA Object" on page 95.

One Listener Per Service Object, Same Partition

If you define multiple service objects as CORBA servers, you can have a listener for each service object and put both the listener and its service object in the same partition, as shown in Figure 1-4. This is the default configuration. That is, if you use the name service or object_to_string to pass the object reference and you never use the configuration dialog, this is the way each service object is configured.

Figure 1-4 An IIOP Client Invoking Methods of iPlanet UDS Service Objects



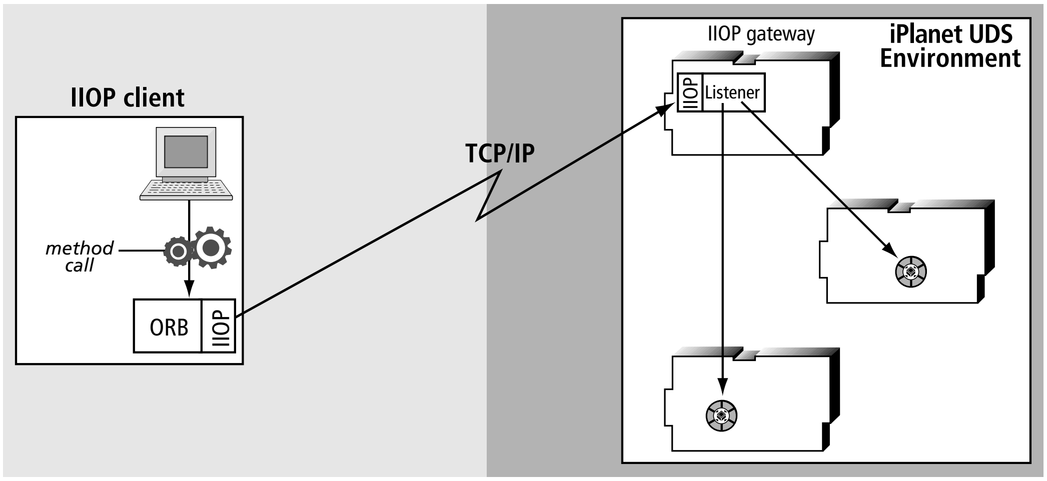
The listener location setting of each listener is `here` (same partition as the service object) and the mode is `forward` (send the request on to the object) for each service object. The forward mode is specified by leaving the Redirect Requests check box blank. See "Using the IIOP Configuration Dialog" on page 48 for more information on these settings.

Forward Mode

If you define multiple service objects as CORBA servers, you can use a single listener for all of them. As shown in Figure 1-5, you can put the listener in its own partition and each service object in a separate partition. You must then specify that the listener forward each request to the appropriate service object in the appropriate partition. You do this by setting the IIOP configuration dialog for each service object as follows:

- Specify the host name and port number of the listener's partition.
- Specify forward mode by leaving the Redirect Requests options box blank.
- Specify Remote for the Listener Location.

Figure 1-5 A Single Listener for all Service Objects in the iPlanet UDS Environment

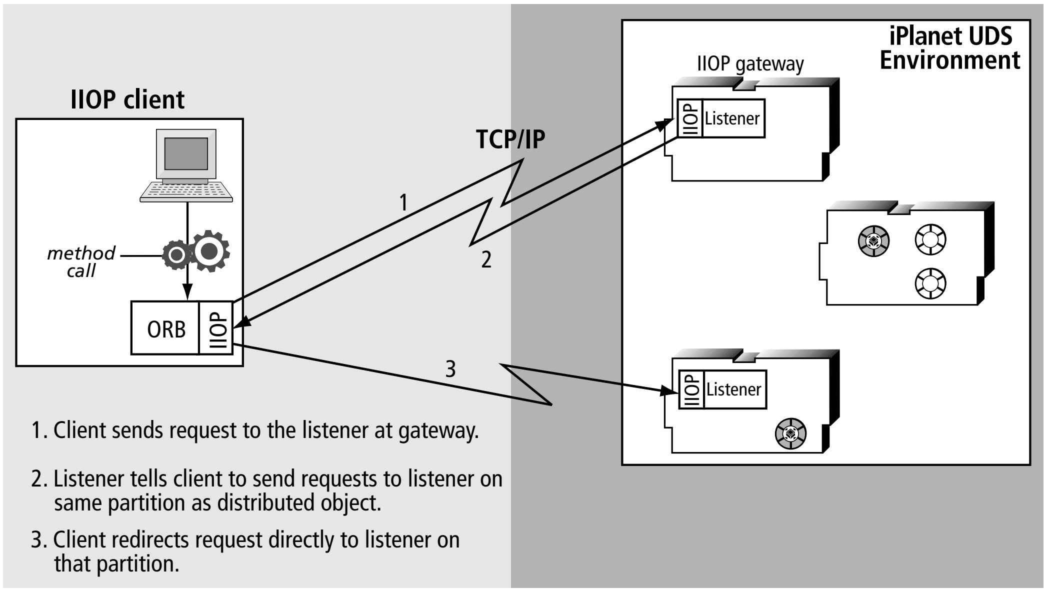


Redirect Mode

In redirect mode, illustrated in Figure 1-6, the listener notifies the caller to send the IIOP request to the partition containing the appropriate CORBA service object. If the service object is in another partition, iPlanet UDS automatically starts a listener for that partition if necessary.

In Figure 1-6, the IIOP gateway application is the listener. The mode is `redirect`. For the service objects in other partitions, the ListenLocation for the listener is `remote`, and the port and host values are those of the IIOP gateway.

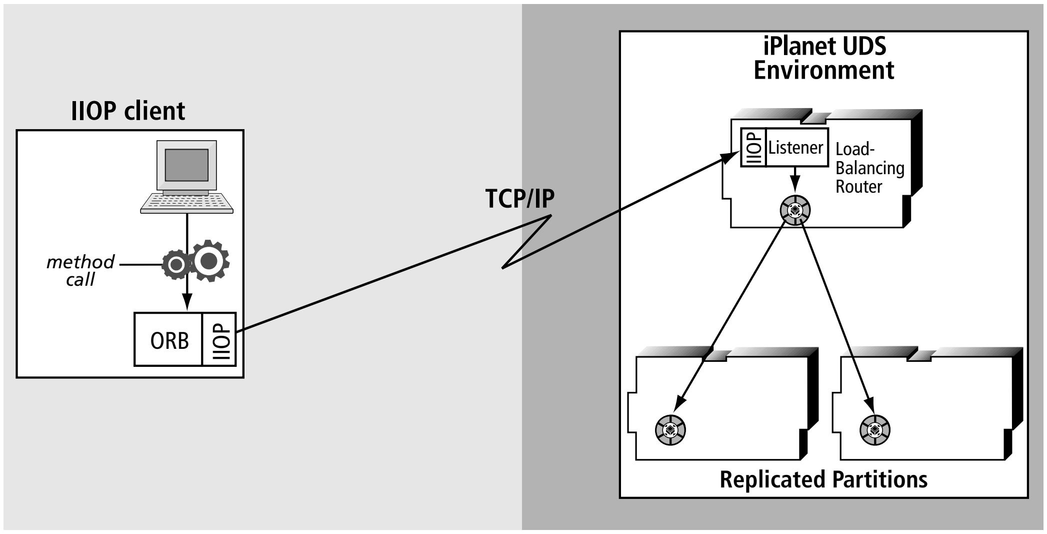
Figure 1-6 Having IIOP Clients Redirect Requests Directly to the Correct Partition


iPlanet UDS Environment
IIOP gateway
IIOP client
TCP/IP
IIOP
Listener
1
2
3
method call
ORB
IIOP
IIOP
Listener
1. Client sends request to the listener at gateway.
2. Listener tells client to send requests to listener on same partition as distributed object.
3. Client redirects request directly to listener on that partition.

Load Balancing

To perform load balancing, the listener must be in the same partition as the router. The router then manages the balancing of requests to the replicated partitions.

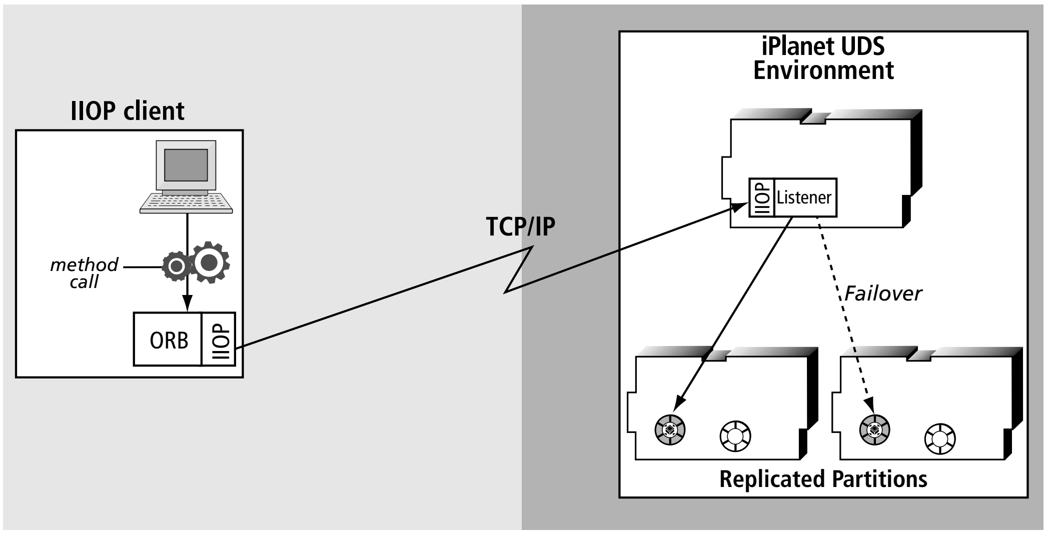
Figure 1-7 Load Balancing with an iPlanet UDS IIOP Server



Failover

Failover only works when the listener is in a different (`remote`) partition from the replicated service objects, as shown in Figure 1-8.

Figure 1-8 Failover with an iPlanet UDS IIOP Server



The remote listener maintains a list of potential failover candidates (server partitions). A single IOR file is generated for this remote listener partition.

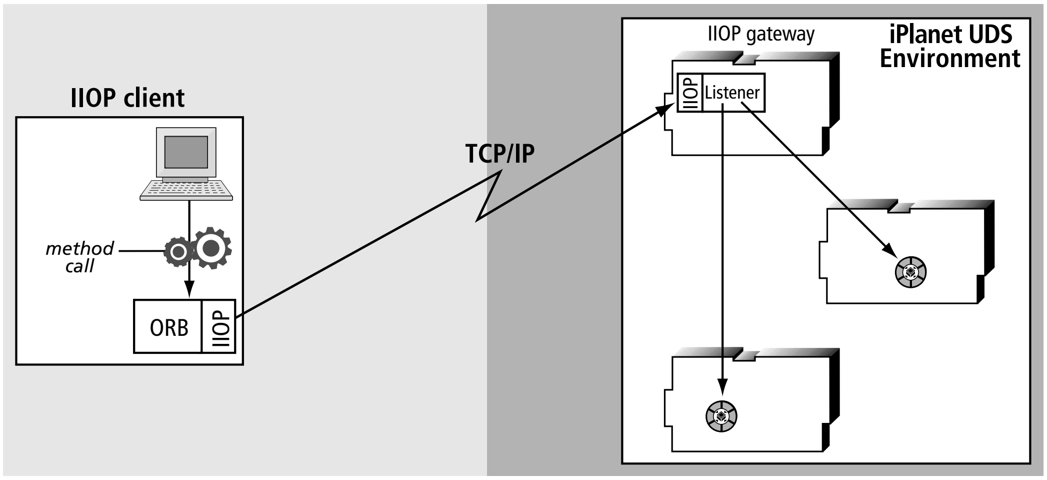
NOTE To get failover support, the listener must not be in the same partition as the server. If the listener is in the same partition for each of the failover partitions, only one of the partitions will be usable by CORBA clients because the same IOR file is used for all the partitions. With only one partition usable, failover will not occur.

Using the iPlanet UDS IIOP Gateway

If you want to have a listener run separately from your application, you can use the *IIOP gateway*, an iPlanet UDS application named iiopgw that starts a listener. You can also use this application when you want to have all IIOP clients send requests to a single listener at a single port.

Using a single port is a way to avoid restrictions imposed by Java security, which permits an external browser running a Java applet to access only the Web server machine. You can have CORBA clients access a single listener that resides on the Web server machine (if necessary, a listener started by an IIOP gateway). iPlanet UDS then routes the requests to other TOOL partitions, as shown in the following figure:

Figure 1-9 Using the IIOP Gateway



Installing the IIOP Gateway Application

You can install the IIOP gateway application by using the Environment Console or Escript.

➤ **To install the IIOP gateway application with the Environment Console**

1. Load the application distribution for the iiopgw application by choosing File > Load Distribution in the Environment Console's Active Environment window and selecting the iiopgw_cl0 application distribution.
2. Put the iiopgw server partition on a node running on the machine where you want to have a listener running.
    a. In the Environment Console's Node Outline view, lock the environment.
    b. Drag or cut-and-paste the server partition to the node where you want the server partition installed.
    c. Unlock the environment.
3. Install the application.

    In the Environment Console's Application Outline view, select the iiopgw_CL0 Application agent and choose Component > Install.

➤ **To install the IIOP gateway application with Escript**

1. Load the application distribution for the iiopgw application by using the following command:

    ```
    escript> LoadDistrib iiopgw cl0
    ```

2. Put the iiopgw server partition on a node running on the machine where you want to have a listener running. Choose the Application agent for the iiopgw_cl0 application, then use the following commands:

    ```
    escript> LockEnv
    escript> FindSub iiopgw_cl0_Part1
    escript> Assign <node_name>
    escript> UnlockEnv
    ```

3. Install the application by choosing the Application agent for iiopgw_CL0 Application agent, then using the `Install` command.

Starting the IIOP Gateway

Typically, the IIOP gateway starts automatically with the startup of a partition that contains a service object marked as needing a remote listener. However, you can also start the IIOP gateway manually.

By default, iPlanet UDS automatically tries to start the IIOP gateway application if a listener is not running where a service object expects one. If the IIOP gateway application is not installed on the node where iPlanet UDS tries to start it, iPlanet UDS raises a DistributedAccessException. You can set the DisableAutoStartGW property to prevent iPlanet UDS from trying to automatically start the IIOP gateway application, as described in "Disableautostartgw Parameter" on page 101.

Setting Up for Automatic Startup of the IIOP Gateway

Before the IIOP gateway can be started automatically, you must define the IIOP configuration parameter by using either the Environment Console or Escript.

➤ **To define the IIOP port configuration parameter in the Environment Console**

1. Open the properties dialog for the installed iiopgw_CL0_Part1 partition on the Installed Partition agent.
2. Enter the `-iiop` flag and the configuration parameters in the Server Arguments field.
3. Click the Close button.
4. Choose Component > Startup to start the partition.

➤ **To define the IIOP port configuration parameter in Escript**

Specify the `-iiop` flag and the parameter by using the Installed Partition agent's `Startup` command, as follows:

```
escript> Startup "-iiop port=2500"
```

Starting an IIOP Gateway Manually

When you start an IIOP gateway manually, you can define the `port` parameter for the IIOP gateway application. To specify this configuration parameter, you need to use the `-iiop` flag defined for the IIOP gateway (iiopgw) application. You cannot define the `IORFile` parameter for an IIOP gateway. If the configuration parameter contains any spaces, you must enclose the entire value in quotation marks.

You can specify the port number where the listener should listen for messages by using the `port` parameter, as described in "Port Parameter" on page 100. For example, you can use the following command. Note that the following command is invoked on one line.

```
ftexec -ftsvr 0 -fi bt:$FORTE_ROOT\userapp\iiopgw\
   cl0\iiopgw1 -iiop "port=2500"
```

Configuring a Service Object to Use the IIOP Gateway

The IIOP gateway application starts a listener on the same node as the gateway (the node that the gateway is installed and running on). Service objects enabled to be CORBA servers can specify that they use this listener by specifying Listener Location = remote, Host = *host_name*, and Port = *port_number*, where the *host_name* and *port_number* correspond to those of the listener started by the IIOP gateway.

For example, if the IIOP gateway is installed on a node on the machine called James and is configured to listen at port 4500, an IIOP server in the iPlanet UDS environment can specify its Listener Location, Host, Port, and IORFile parameters as follows:

Listener Location = remote, Host = James, Port = 4500, IORFile = (name = BankSvc)

In this case, the service object defines its listener as the one started by the IIOP gateway application. This listener then routes messages for the service object to the partition containing the service object.

For more information on configuring service objects, see "Using the IIOP Configuration Dialog" on page 48.

Building a CORBA-Enabled Application

This section describes the steps required to write and build a CORBA application.

Create the IDL Interface Definition

1. Get the IDL text file that describes the interface of the CORBA object whose methods you want to invoke.
2. Use the appropriate tool or compiler to generate the files required by CORBA: these normally include the implementation version of the IDL interface, the client stub, the server skeleton, and one or more helper classes. For example, use the idlj (jdk 1.3) compiler to generate files for a CORBA object accessed from Java; use the corbagen tool to generate files for a CORBA object accessed from TOOL. The table below lists the IDL compilers supported in the current version of iPlanet UDS:

Compiler/Tool	**Description**
Sun idlj.exe	Maps IDL to Java. Shipped with JDK 1.3.
corbagen.exe	Maps IDL to TOOL. Shipped with iPlanet UDS. You can also use the Plan > Import IDL command to generate and import an IDL file. You may prefer to use the corbagen tool if you are using scripts.
Iona OrbixWeb 3.2 idlj.exe	Maps IDL to Java.
Inprise VisiBroker 4.0 idl2java.exe	Maps IDL to Java.

If you use the IIOP configuration dialog to configure a TOOL server, the IDL file is automatically generated for you.

Implement the Client

A client that wants to use a distributed CORBA object must do the following:

1. Include the required libraries and the files generated by the IDL compiler.
2. Get a reference to the object implementation from an IOR file or from the naming service.
3. Invoke the method.

For code examples that illustrate these steps, see "Working with CORBA Clients" on page 61.

Implement the Server

A server containing a CORBA object that can be accessed by a client must:

1. Include the required libraries and the files generated by the IDL compiler.
2. Provide a reference to the object implementation using an IOR file or the naming service.
3. Implement the CORBA object that contains the methods whose interface are given in the IDL file.

If you use the IIOP configuration dialog to configure a TOOL server, the IDL file is automatically generated for you.

The implementation of these steps differs depending on the server's source language and on the mechanism used to pass the object reference to the client. For code examples that illustrate these steps, see "CORBA Servers" on page 41.

Build and Run the Application

➤ **To build an application that supports the use of CORBA objects**

1. Use the appropriate tool or compiler on the IDL text file to generate the files required for CORBA support.
2. Compile and link the source files, including the files generated in Step 1 for both the client and the server.
3. Make a distribution. This generates the IDL file if the object implementation was configured using the IIOP configuration dialog. (Not applicable for non-TOOL applications.)
4. If you are using a naming service, make sure the naming service is running.
5. Start the server.
6. Start the client in a different shell or on a different machine from the server.

The first two steps may be completed in different ways, depending on the development environment and the tools you use to build your application.

Chapter 2

CORBA Servers

This chapter begins with an overview of CORBA servers and then examines a Java and TOOL server implementation as examples. You should read this chapter if you need to create, build, test, and deploy a CORBA server.

You need to read the chapter "Interoperating With CORBA Objects" on page 19 before you read this chapter.

Servers

A CORBA server is an application with at least one object that can be accessed by a CORBA client. The client accesses the external object through an object reference that it obtains either through a naming service or through an IOR file. The client developer must also be able to access the IDL file that defines the interface to the external object, in order to generate the stub files required for the CORBA communication infrastructure.

In general a CORBA server, independently of its source language, must do the following:

1. Implement the server that is to be called by CORBA clients.
2. Create an IDL file that describes the server interface
3. Document the location of the IDL file for the client.
4. Make an object reference available to the client.

The first two requirements can be reversed: the server can be implemented based on a given IDL file.

The TOOL and Java examples given in this chapter show two possible server implementations.

Passing the Object Reference to the Client

There are a number of ways to advertise the object reference or to pass it to a client:

- Advertise the object in a CORBA name service.

  This is the preferred method. In the Initialize method of your service object, you need to specify the name for the service object (bind a name to the object reference). The client can then ask the naming service for the reference to the object associated with that name. For an example, see "Implementing the TOOL Server" on page 45 and "Implementing the Java Server" on page 57.

- Use the ORB function `object_to_string`.

  You can call the `object_to_string` function to create a string, which the client can then turn back to an object reference by using the `string_to_object` function. It is up to you to make that string available to the client, typically by writing it to a file that can be accessed by the client.

- Use the service object IIOP configuration dialog.

  Specifying the name of an IOR file in this dialog causes the underlying runtime code to create a file and write the object reference to it. The file is then stored in the directory you specify or in the `FORTE_ROOT/etc/iiopior` directory.

Please note that the IIOP configuration dialog is also used to configure the listener. It is entirely possible to advertise the CORBA object in a name service and to use the configuration dialog to configure the listener if you want to use an IIOP gateway. In this case, you can specify a dummy name for the IOR file and ignore the file.

Accessing Distributed Objects

If the CORBA object returns object references that allow the client to call methods on other server objects, the IDL file must contain IDL declarations for these objects as well, just as it does for the original object implementation. This is true both for Java and TOOL servers.

Clients can access any distributed object in the iPlanet UDS environment for which the following conditions are true:

- The distributed reference to the object can be returned from another object that is accessible through IIOP.

For example, a service object has a method that returns a reference to a distributed object. Your IIOP client can invoke that method to get the reference to the distributed object, then use that reference to directly invoke methods of the distributed object.

- At least one method for the distributed object can be exported as IDL.

  The parameters for the methods of distributed objects must conform to the same rules as those for service objects, as described in "Parameter and Return Values" on page 85.

- It is strongly recommended that all distributed objects inherit from CorbaObjectImpl, or that they implement the interface CorbaObject from the CORBA system project.

  If your distributed objects don't do either of these, you must ensure that the distributed objects are anchored, which means that they reside in one partition, and other partitions access the object using distributed references. Because a class definition can specify that its objects are distributed, but not require that all objects of that class be anchored, it is up to the TOOL programmer to set the IsAnchored property for each distributed object of the class to TRUE. (See the *iPlanet UDS Programming Guide* for a thorough discussion of distributed objects.)

  If the IDL file is generated automatically, iPlanet UDS generates IDL for any class used by the service object that specifies Distributed = Allowed, *even if the objects are not anchored by defaul*t. Because IIOP clients cannot access objects of these classes unless the objects are anchored, you must ensure that these objects are actually anchored when IIOP clients attempt to access them. Otherwise, the client will produce runtime errors.

Developing and Building Servers

TOOL server writers have a number of options to choose from in developing and building the server, depending on the following factors:

- Whether they want the implementing object to be a service object.
- Whether they want the IDL file to be generated automatically.
- Whether they want the IOR file to be generated automatically.
- How they want to pass the object reference to the client.

Object or Service Object The object implementation does not have to be a service object. If it is not, you cannot use the IIOP configuration dialog and therefore cannot have the IDL file and the IOR file automatically generated. You must write them yourself. You also cannot specify the location of the listener, but must use the default configuration. For a description of the default configuration, see "One Listener Per Service Object, Same Partition" on page 29. This may change in a later release.

IDL File Generation You can either develop the server from scratch and have the system generate the IDL file automatically at partitioning time or you can start with an existing IDL file which you compile and import into the UDS environment. If you use the second method, you need to specify the imported project as a supplier plan to your server project.

If you have used the IIOP configuration dialog in defining the properties of your service object, an IDL file is automatically generated. If you are not interested in this IDL file, you can discard this file and simply make the external IDL file a supplier to your project as described in the next section, "Working with TOOL Servers."

Note that by starting the server from an existing IDL file, you retain more control over the IDL file because you can keep implementation methods, which the client will never call, private to the server.

You should avoid making changes to a project that was imported from an IDL file. You should also avoid exporting and reimporting it as a TOOL project.

Naming Service or IOR file If you use the naming service to register CORBA objects, you can still use the IIOP Configuration dialog to specify the location of the listener. In this case, an IOR file is still generated, but it can be ignored since it is not needed.

For more information about TOOL CORBA server architecture, see "TOOL Server Design" on page 27.

Working with TOOL Servers

The following sections explain the steps you must take to create a TOOL server. Note that in this case—where the CORBA object is a service object and the IIOP configuration dialog is used, generating the IDL file is part of making a distribution, as shown in Step 9 on page 47. As mentioned earlier, the object implementation does not have to be a service object. However, this is likely to be the most common case, and therefore, a service object is used in the following example.

➤ **To create a project for the server and specify the suppliers for your project**

1. Create a project for the server implementation by choosing Plan > New Project from the Repository Workshop and specifying the name of your project, for example, TestServer.
2. To specify the suppliers to your project, open the Project Workshop and choose File > Supplier Plans. Specify the CosNaming library and the CORBA library as suppliers.
3. If you have used an external IDL file, you must do one of the following:

- Use the Plan > Import IDL command to compile and import the IDL file. The imported project(s) will have the same name as the module(s) contained in the IDL source file. Add the project(s) as supplier to your server project.
- Use the corbagen utility to compile the IDL source file, and import the resulting .pex file into the iPlanet UDS Workshop. The imported project(s) will have the same name as the module(s) contained in the IDL source file. Add the project(s) as supplier to your server project.

  You should avoid making changes to a project that was imported from an IDL file. You should also avoid exporting and reimporting it as a TOOL project.

Implementing the TOOL Server

As opposed to creating a Java server (which includes setup code as well as the object implementation), when you create the TOOL server, the setup work is part of the initialization method for the service object.

➤ **To create the service object using the CORBA name service**

1. Create a distributed class, for example TestClass, in your server project (TestServer).
2. Create the service object TestClassSO for TestClass.
3. Specify the host where the name service is running.
4. Make sure that the class for your service object implements the interface Framework.SOInitializer.
5. Add a method Initialize() to your service object with the following signature:

```
Initialize(): integer
```

6. Provide the following implementation for the Initialize method:

```
theOrb : Corba.ORB = Corba.ORB();
args : Array of TextData = new;
//Specify port where name service is running.
args.AppendRow(TextData(Value = '-ORBInitialPort 1050'));
theOrb.Initialize(args);

nameArray : Array of CosNaming.NameComponent = new;
nameComp : CosNaming.NameComponent = new;

nameServiceObj : Corba.CorbaObject =
      theOrb.resolve_initial_references('NameService');
// narrow
nameServiceObj = theOrb.narrow(nameServiceObj,
      CosNaming.NamingContext);
// typecast
nameService : CosNaming.NamingContext =
      CosNaming.NamingContext(nameServiceObj);

nameComp.id = ‘ToolTestServer’
nameComp.kind = ‘’;
nameArray.AppendRow(nameComp);

nameService.rebind(nameArray,self);

return 0;
```

Note that this code assumes that your name service is an interoperable name service. This works for the Sun JDK. For VisiBroker and Iona OrbixWeb, see Technical Note 12500.

7. Add a method TestServer.HelloMethod() to your service object with the following signature:

```
HelloMethod(param : string)
```

8. Provide the following implementation for HelloMethod:

```
task.Lgr.PutLine(‘Entering TestServer.HelloMethod’);

task.Lgr.PutLine(param);

task.Lgr.PutLine(‘Leaving TestServer.HelloMethod’);
```

9. Make a distribution.

If you need to configure the listener, generate an IDL file, or specify the name of an IOR file, you will need to bring up the IIOP Configuration Dialog. See "Using the IIOP Configuration Dialog" on page 48 for directions on how to display this dialog.

Note that using this dialog always causes an IDL file to be generated automatically. The file is stored in the following directory:

```
appdist\testenv\testserv\c10\generic\testse1
```

If you are basing your server on an existing IDL file, you can ignore this file.

10. Use the following command for the JDK 1.3 to generate the required Java stub code:

```
idlj -fclient corba1.idl
```

Use the equivalent command if you use another ORB.

Using an IOR File to Pass an Object Reference

When you use the IIOP configuration dialog to configure the object implementation, an IOR file is automatically generated for you. If you do not use the configuration dialog and are not using a name service, you can use code like the following in the object's SOInitializer.Initialize() method to get the object reference into a form that you can pass to the client.

```
// theOrb and portableFileName : String are assumed to have been
// defined
iorString : String = theOrb.object_to_string(obj);
f: File = new;
f.SetPortableName(portableFileName);
f.Open;
f.Write(iorString);
f.Close;

// Catch I/O exceptions here & make sure to close file
```

The IIOP configuration dialog is described in the next section.

Using the IIOP Configuration Dialog

You use this dialog to do any one of the following:

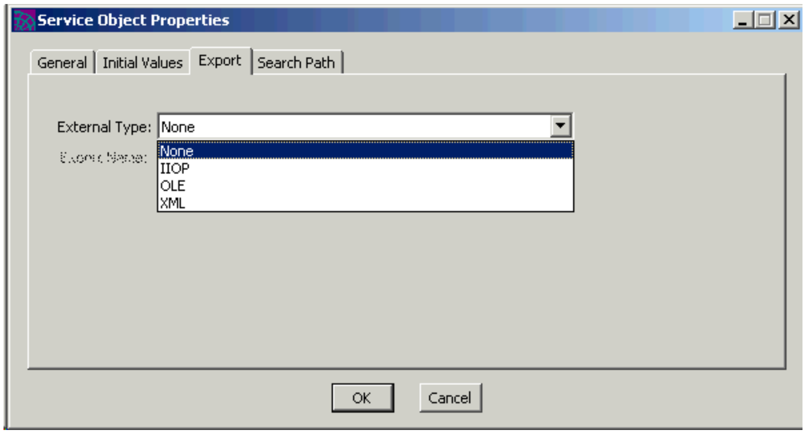
- Automatically generate the IDL file
- Configure the listener if you use an IIOP gateway
- Automatically generate an IOR file

If you do not need to do any of the above, you do not need to use this dialog.

You can also use the Fscript command `SetServiceEOSInfo` to specify the information set in this dialog. For directions on how to use this command, see page 51.

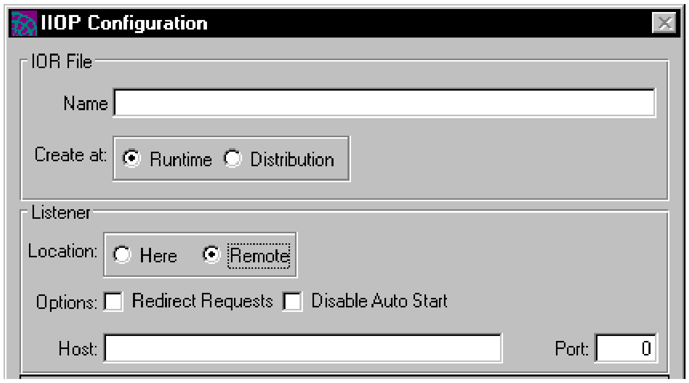
➤ **To set IIOP configuration using a dialog**

1. In the Repository Workshop, select the main project for the iPlanet UDS application and click the Partition Workshop icon.
2. In the Partition Workshop, expand the server partition to display the service object, then double-click the service object to open its Service Object Properties dialog.
3. Select the Export tab page and choose IIOP for the External Type, then click OK to display the IIOP Configuration dialog.



External Type: None

4. In the IIOP Configuration dialog, specify the service object's IIOP configuration settings.


IIOP Configuration

IOR File

Name

Create at: [✓ Runtime] Distribution

Listener

Location: Here [✓ Remote]

Options: ☐ Redirect Requests ☐ Disable Auto Start

Host:

Port: 0

The settings are described in the following list:

- IOR File settings

Field	Description
Name	Specifies the name and location of the IOR file. If you do not specify a path, the file is put in FORTE_ROOT/etc/iiopior directory.
Create at	Specifies when the IOR File is created. By default, the IOR File is created at runtime. `Runtime` specifies that the IOR File is created when the service object is started. This setting is the default and gives you the most flexibility. `Distribution` specifies that the IOR File is created at the time the application distribution is made. If you choose `distribution`, your development environment must be the same environment as your deployment environment.

- Listener settings (for more information, see "TOOL Server Design" on page 27)

Field	Description
Location	Specifies whether the listener is in the same partition (here) or on another partition (remote). `Here` specifies that a listener starts in the same partition as the service object. `Remote` specifies that a listener in another partition or an IIOP gateway receives requests for this service object. If you choose this option, you must also specify the listener's Host and Port fields.
Options	Specifies how IIOP requests get routed from an IIOP client to the iPlanet UDS distributed object in another partition. By default, a listener receives each request from a client and forwards these requests to the service object. `Redirect Requests` specifies that the IIOP client send requests directly to a listener on the partition containing the distributed object. The `Disable Auto Start` toggle is available only if you have chosen to use a remote listener (Remote radio button). You can check this option to specify that the IIOP gateway not be automatically started if it is not already running.
Host	Specifies the name (or the IP address) of the host machine on which the listener is to run. This host name is included in the IOR file. You can only specify a host name if the listener location is Remote. If the IOR string is to be created at distribution time and you do not specify a host name, then the name of the host on which the application distribution is made is put into the IOR string. This default might not be appropriate when you deploy your application in a runtime environment. Generally, when you create an IOR string at distribution time and you have a remote node that uses IIOP, you must specify the host name of that remote node. If the IOR string is to be created at runtime and you do not specify a host name, the name put in the IOR string is name of the host on which the partition that contains the service object is running.

Field	Description
Port	Specifies the port number for a listener to which the IIOP client can send requests. The port number is included in the generated IOR file. You must specify a port number when the IOR file is created at distribution time or if you have specified a remote listener. You do not have to specify a port number when the listener is `here` and the IOR file is generated at `runtime` because the operating system can assign a port number at runtime, and iPlanet UDS can include that port number in the IOR file.

5. Click the OK button.

For more information about specifying service object properties, see *A Guide to the iPlanet UDS Workshops*.

➤ To set IIOP configuration using the SetServiceEOSInfo command

1. Start Fscript.
2. Open the repository. Make the deployment environment the current environment, and make the main project for the application containing this service object the current plan. For example:

```
fscript> Open

fscript> FindEnv MyEnvironment

fscript> FindPlan MainProject
```

3. Partition the application with the `Partition` command.

4. Enter the `SetServiceEOSInfo` command by using the following syntax:

```
SetServiceEOSInfo service_object_name iiop
   " IORFile = (name=iorfile_name [, runtime | dist])
   [, listenlocation = here | remote] [, forward | redirect]
   [, host = host_name] [, port = port_number]
   [, disableautostartgw] [,outbound]"
```

service_object_name is the name of the service object that you want to make available to IIOP clients. If the current project contains the service object, you can specify just the name of the service object; otherwise, *service_object_name* should specify the project name and the service object name, separated by a period.

The following example marks the BankServer service object as an IIOP server. Note that the following command is invoked on one line.

```
fscript> SetServiceEOSInfo BankServices.BankServer iiop
   "iorfile=(name=BankServ.ior)"
```

In this example, BankServices is the name of the project, and BankServer is the name of the service object. BankServ.ior is the file name for the IOR file, which is placed in the FORTE_ROOT/etc/iiopior/ directory at runtime.

Note that support for outbound IIOP service objects (UDS is a client) is deprecated as of the current release.

For more information about using Fscript to specify IIOP configuration settings, see Appendix B, "Using Fscript to Configure CORBA Servers" on page 95.

For more information about Fscript, see the *Fscript Reference Guide*.

Testing, Making the Distribution, and Installing

This section provides an overview of how you can test and deploy your TOOL application for use as an IIOP-IDL server.

Testing Your Application if You Are using IOR Files

You can test run your iPlanet UDS application with IIOP clients by using the iPlanet UDS Partition Workshop or Fscript if you specify that the IOR files are generated at runtime. Even if you want to have the IOR file for the application generated at distribution time, you need to specify that the IOR files be generated at runtime to test run the application within the Partition Workshop or Fscript. You can then change the configuration parameter back to distributed after you have finished your test runs.

For details about performing a test run in the Partition workshop, see *A Guide to the iPlanet UDS Workshops*. For details about performing a test run in Fscript, see *Fscript Reference Guide*.

Making a Distribution

Make a distribution using the Partition Workshop or the Fscript `MakeAppDistrib` command. Making a distribution generates IDL and, optionally, IOR files, which provide the information needed to access iPlanet UDS service objects as IIOP servers, as well as the usual files for the iPlanet UDS partition.

No need to compile for IIOP iPlanet UDS does not produce any server stubs that require compiling, so you do not need to use the Fcompile utility or the auto-compile facility to enable IIOP clients to access the iPlanet UDS service object. If the partition containing the service object is marked as compiled, then you can compile this partition as usual, using the Fcompile utility or the auto-compile facility, as described in *A Guide to the iPlanet UDS Workshops*.

When you make a distribution that includes one or more partitions that contain service objects that are marked as IIOP servers, iPlanet UDS generates an IDL file for each partition called corba#.idl. Each IDL file contains the IDL for all classes in the partition.

The IDL file for a partition is placed in the following directory path when you make a distribution:

`FORTE_ROOT/appdist/`*environment_id*`/`*application_id*`/cl#/generic/`*partition_id*`/`

If you install the application, the IDL file is placed in the same directory as the other files for the partition:

`FORTE_ROOT/userapp/`*application_id*`/cl#/`

If more than one service object in a partition is marked as an IIOP server, the IDL for all the distributed objects is placed in the same corba#.idl file for the partition.

If you also select auto-install, making the distribution also installs the IDL files on the appropriate nodes in the development environment, according to the configuration you specified when you partitioned your TOOL application. Auto-install is usually a testing convenience. Unless your deployment environment and development environment are identical, do not auto-install.

NOTE If you chose to create the IOR file at `distribution`, your deployment and development environments must be the same environment. For information on choosing when to create the IOR file, see the description of the Create At field on page 49.

If you specified `runtime` as an IOR File configuration setting, iPlanet UDS generates the IOR string when you start the partition containing the service object. If you are deploying the application, runtime is usually the best choice.

For information about making a distribution using the Partition Workshop see *A Guide to the iPlanet UDS Workshops*. For information about the Fscript `MakeAppDistrib` command, see the *Fscript Reference Guide*.

Installing the iPlanet UDS Application

If you used auto-install, you are done with this procedure. Otherwise, use the Environment Console or Escript to install the application containing the iPlanet UDS service object using standard iPlanet UDS installation procedures. For more information about installing applications in iPlanet UDS, see the *iPlanet UDS System Management Guide*.

Documenting the IIOP Server

When iPlanet UDS generates the IDL for a partition containing one or more service objects, iPlanet UDS generates the IDL only for a service object's classes. The user will not be able to determine from the IDL which classes are the classes for the service objects. Therefore, along with the methods, attributes, and exceptions for each class, you also need to document for each service object:

- the service object's class
- the name and location of its IOR file
- when the IOR files are generated
- the partition containing the service object

Working with Java Servers

The following sections explain how you write and build a Java server.

If you are writing a Java CORBA server, you must start with the interface given in the IDL file. In TOOL this is optional. The Sun JDK and other Java ORB vendors provide a tool to translate IDL to Java, but not to translate Java to IDL. TOOL on the other hand can create IDL from an existing server implementation.

Remember that you must first compile the IDL interface file to provide the mapping and support required by the Java server. You use the same IDL source text file for the server as you did for the client.

```
module HelloModule
{
    interface HelloInterface
    {
      void HelloMethod(in string param);
    };
};
```

➤ To compile the file for the Java server

1. Start a command-line shell and run the compiler on the idl source file. (For information on flag settings for your compiler, please consult the documentation provided). The following sample command compiles the Hello.idl source file using the JDK 1.3

   ```
   idlj -fserver hello.idl
   ```

   This generates up to six files in a subdirectory that has the same name as the module, in this case, helloModule. The table below describes the contents of these files:

File name	Contents
`_HelloInterfaceImplBase.java`	An abstract class that functions as the server skeleton, providing basic CORBA functionality for the server. It is used to implement the HelloServer.java interface. This is the abstract class from which the server implementation inherits.
`_HelloInterfaceStub.java`	The client stub that provides CORBA functionality for the client. It implements the HelloServer.java interface.
`HelloInterface.java`	The Java version of the .idl file. It contains the single method, HelloMethod. The Hello.java interface extends org.omg.CORBA.Object, providing standard CORBA object functionality as well.
`HelloInterfaceHelper.java`	A class that provides additional functionality, primarily the narrow() method required to convert CORBA object references to their proper type.
`HelloInterfaceHolder.java`	A class that holds a public instance member of type Hello. It provides operations for out and inout arguments, which CORBA has but which do not map readily to Java's semantics.
`HelloInterfaceOperations.java`	Base class to HelloInterface (JDK 1.3 only).

Implementing the Java Server

The server implementation needs to contain two classes: a main class that does the setup work and a class that provides the object implementation. The following code shows how you do this in Java.

```
// import required libraries
import java.io.*;
import java.util.*
import java.Properties;
import org.omg.CORBA.*
import org.omg.CosNaming // if using the name service

// the object implementation
class myHello extends helloModule._HelloInterfaceImplBase
{
  public void HelloMethod(String param1)
  {
    System.out.println("In server's HelloMethod");
    System.out.println(param1);
  }
};

// The server implementation
public class HelloSvr
{
  public static void main(String[] args)
// set default value for host name and port number if these
// are not passed from the command line that starts the server
  {
  try
  {
    String orbArgs[];
    if (args.length > 0)
    {
      orbArgs = args;
    }
    else
    {
      String defaultArgs[] = {"-ORBInitialPort", "900"};
      orbArgs = defaultArgs;
    }
// initialize ORB
    org.omgCORBA.ORB orb = org.omg.CORBA.ORB.init(orbArgs, null);
// get reference to name service and narrow it to desired type
    org.omg.CORBA.Object objNameService =
          orb.resolve_initial_references("NameService");
    NamingContext ctx =
NamingContextHelper.narrow(objNameService);

// initialize object instance and register it with the ORB
    myHello svrMine = new myHello ();
    orb.connect(svrMine);
// advertise (bind) the object reference in the naming context
    NameComponent nc1 = new NameComponent("MyServer", "text");
    NameComponent[] name1 = {nc1};
    ctx.rebind(name1, svrMine);

// listen for method invocation
    try
    {
      java.lang.Object sync = new java.lang.Object();
      synchronized (sync)
        {
          sync.wait();
        }
      }
// error handling
    catch (InterruptedException ie)
    { System.out.println("Hello Servers shutting down.");}
    catch(org.omg.CORBA.UserException e)
    { System.err.println(e);}
    catch(org.omg.CORBA.SystemException e)
    {System.err.println(e);}
  }
}
```

Using an IOR File to Pass an Object Reference

The previous code sample uses the name service to make object references available to the client. It registers the object with the name service; the client code can then query that service to obtain the object reference.

However, you also have the alternative of passing the object reference to the client by way of an IOR (interoperable object reference) file. In order to do this, the server's main function must include code like the following:

```
//in server main
myHello svrIOR = new myHello("FROM_IOR");

//register object with ORB
orb.connect(svrIOR);

//use ORB method to convert object reference to string
String ior = orb.object_to_string(svrIOR);

//use helper function to write string to file
PutIOR(ior, "d:/mydir/hello.ior");
```

The PutIOR helper function, which you use to write the string to a file, is shown in the following code sample. You need to include this method in your Java server code.

```
public static void PutIOR(String ior, String fileName)
{
  DataOutputStream output = null;
  try
  {
    FileOutputStream x = new FileOutputStream(fileName);
    output = new DataOutputStream(x);
  }
  catch(IOException ex)
  {
    System.err.println(ex);
    return;
  }
  try
  {
    output.writeBytes(ior);
    output.close();
  }
  catch(IOException ex)
  {
    System.err.printlen(ex);
  }
}
```

Building the Server

➤ **To build the Java server**

1. If you have not already done so, run the idlj compiler on the IDL file to create the stubs, skeletons, and other supporting files.

   ```
   idlj hello.idl
   ```

2. Compile the .java files, including the files generated by the idlj compiler. The code sample below assumes that all the files (source and generated files) are in the same directory.

   ```
   javac *.java
   ```

3. If you are using a name service, start it now:

   ```
   tnameserv -ORBInitialPort 1050&
   ```

4. Start the Java server.

   ```
   java HelloSvr -ORBInitialPort 1050
   ```

Chapter 3

Working with CORBA Clients

This chapter begins with an overview of CORBA clients and then examines a Java and TOOL client implementation as examples. You should read this chapter if you need to create, build, test, and deploy a CORBA client.

You need to read the chapter "Interoperating With CORBA Objects" on page 19 before you read this chapter.

Client Overview

This section provides general guidance for writing and running clients that interact with external object implementations using the ORB infrastructure.

For additional examples of IIOP clients for iPlanet UDS IIOP servers with the ORB products VisiBroker for Java or OrbixWeb, see Technical Note 12500.

Files Needed by IIOP Clients

In order to call an external object implementation transparently, the client needs access to the following files:

- IDL files: these text files specify the interface to the object implementation. The ORB product you are using provides an IDL compiler that converts the interface defined in IDL to the client's source language.

  The IDL compiler generates several output files. The stub file is specifically needed by the client both to enable language transparency and because the stub contains generated code that handles the marshalling of method invocations.

The server creator is responsible for generating the IDL files and making them available to the client.

- IOR files: these text files contain strings that specify references to external object implementations. The client uses the IOR string to bind to the object implementation at runtime. *If the client gets an object reference using a naming service, no IOR file is needed*.

  If the CORBA object is configured using the IIOP Configuration dialog, the IOR file is automatically generated either when the distribution is made or at runtime. If the IIOP Configuration dialog is not used, the server creator must create the IOR file manually using the ORB method object_to_string. Either way, the server creator is also responsible for documenting the file's use, explaining how to locate the file and, if applicable, when it is generated.

  The client must access the IOR file, read the IOR strings from the IOR file, and use the ORB's string_to_object method to turn these strings into object references. The client can then use the object references to call server methods.

Locating TOOL IDL files

If the TOOL object implementation has been configured using the IIOP Configuration dialog, iPlanet UDS generates an IDL file for each partition that contains a service object marked as an IIOP server. Each IDL file is named corba#.idl, where # is the number of the partition. The IDL file for a partition is placed in the following directory path when you make a distribution:

FORTE_ROOT/`appdist`/*environment_id*/*application_id*/`cl#`/`generic`/*partition_id*/

If you install the application, the IDL files are placed in the same directory as the other files for the partition:

FORTE_ROOT/`userapp`/*application_id*/`cl#`/

If more than one service object in a partition is marked as an IIOP server, the IDL for all the distributed objects are placed in the same corba#.idl file for the partition.

iPlanet UDS also provides IDL files for the iPlanet UDS libraries in the FORTE_ROOT/install/inc/idl/ directory, as shown in the table below:

IDL File Name	iPlanet UDS Library
`displayp.idl`	Display library
`framewor.idl`	Framework library
`genericd.idl`	GenericDBMS library
`systemmo.idl`	SystemMonitor library

Generating Client Stubs from IDL

Having obtained the IDL file, the client must compile the file to create the required client stubs. The ORB product for the client provides an IDL compiler that converts the IDL files to another common language, such as Java.

➤ **To generate the client stubs from the IDL file provided by the IIOP server**

1. Copy the IDL files to your working directory.

   If the generated IDL files include the IDL files for iPlanet UDS libraries, such as framewor.idl or displayp.idl, then you must also copy these IDL files to your working directory.

2. Run the IDL compiler to generate the client stubs.

You can then write IIOP client applications that use the generated classes to interact with the iPlanet UDS IIOP server.

See the documentation for your ORB product to determine the following information:

- how the IDL maps to the client stubs generated by the ORB product's IDL compiler

  This information includes descriptions of how data types, classes, methods, exceptions, and so forth are represented in the client stub files. The appendix "IDL and TOOL" on page 81 describes the mapping that is done between IDL types and TOOL types, and between TOOL types and IDL types.

- how to write IIOP client applications that use the client stubs
- how to run IIOP client applications that use the client stubs

Writing an IIOP Client Application

An IIOP client that accesses a server can be written using the same approach that you would use for any other client. That is, it is a normal client application with some added code whose purpose is to obtain references to object implementations. Having obtained these references, the client can then use these to call methods implemented by the external object. There are two ways to get object references: using the naming service or using an IOR file.

If you are using an ORB product to write your IIOP clients, its documentation describes how to write, compile, link, and distribute IIOP client applications.

Using the Naming Service

If the server registers the object implementation with the Naming Service, the client must include code that does the following:

- initializes the client-side ORB
- gets a reference to the naming service
- obtains an object reference for the object whose methods the client wants to invoke, and invokes those methods

For code samples showing how the preceding steps are implemented, see "Implementing the TOOL Client" on page 75 and "Implementing the Java Client" on page 70.

Reading the IOR File

If the server uses an IOR string to pass the reference to the object implementation to the client, the client must open the IOR file, read the IOR string, and use the ORB method `string_to_object` to convert the string to an object reference.

See "Implementing the TOOL Client" on page 75 and "Implementing the Java Client" on page 70 for information about how you do this.

Check the documentation for your ORB development product for more information about how to obtain an object reference from an IOR string.

Accessing Distributed Objects

Clients can access any distributed object in the iPlanet UDS environment for which certain conditions have been met. These are described in "Accessing Distributed Objects" on page 42. For example, suppose the service object has a method that returns a reference to a distributed object. Your IIOP client can invoke that method to get the reference to the distributed object, then use that reference to directly invoke the distributed object's methods.

The following example shows a Java client accessing attributes of a TOOL distributed object:

```
-- ForteSO is a reference to a Forte service object.
-- Flight.FlightDetails is the class for a Forte distributed
object.
-- ForteSO.QueryFlight() returns a distributed reference to
-- a Flight.FlightDetails object.
-- DepartFrom, DepartTime, ArriveAt, and ArriveTime are
attributes
```

```
-- of the FlightDetails object.
    -- Get distributed object reference
Flight.FlightDetails f = FlightSO.QueryFlight(flightNumber);
    -- Call object's method
String origin = f.DepartFrom();
String departTime = f.DepartTime();
String destination = f.ArriveAt();
String arriveTime = f.ArriveTime();
```

If the server is configured with the IIOP Configuration dialog, iPlanet UDS automatically generates an IDL file for any class used by the service object for which the extended property Distributed is set to Allowed, even if objects of the class are not anchored by default. All methods that conform to the restrictions described in "Parameter and Return Values" on page 85 are included in the generated IDL. Therefore, when you get the reference to the distributed object, you can use its methods.

NOTE Clients cannot access TOOL objects unless they are distributed objects, which means that they must be anchored. If your client gets errors when accessing a TOOL object, the object might not actually be anchored at runtime, even though the object's class definition allows the objects to be distributed. See "Accessing Distributed Objects" on page 42 for more information about this issue.

iPlanet UDS maintains information about references to TOOL distributed objects, including references by IIOP clients. The IIOP client does not need to explicitly disconnect from a TOOL distributed object after it is finished with it.

For a more detailed description about how to access iPlanet UDS distributed objects using Visigenic VisiBroker, see Technical Note 11152. For information about accessing distributed objects using Iona OrbixWeb, see Technical Note 11153.

Interpreting Exception Information

When iPlanet UDS generates an IDL declaration for a method that raises an exception, the IDL equivalent of the method signature by default raises either a GenericException or a UserException. These exceptions contain an ErrorDesc_struct that is defined in the framewor.idl file.

The ErrorDesc_struct contains the following members:

Member Declaration	**Description**
string *ErrorText*;	Message attribute set for the raised iPlanet UDS exception.
string *ReasonCode*;	ReasonCode attribute set for the raised iPlanet UDS exception.
string *Severity*;	Severity attribute set for the raised iPlanet UDS exception.
string *ClassName*;	Name of the class type of the raised iPlanet UDS exception.
i4 *SetNumber*;	SetNumber attribute set for the raised iPlanet UDS exception. Number of the message set that contains the message associated with this exception.
i4 *MsgNumber*;	MsgNumber attribute set for the raised iPlanet UDS exception. Number of the message in the message set associated with this exception.
string *DetectingMethod*;	Name of the method that raised the exception. The DetectingMethod attribute set for the raised iPlanet UDS exception.
i4 *MethodLocation*;	Line number within the method where the exception was raised. The MethodLocation attribute set for the raised iPlanet UDS exception.
string *TOOLClass*;	TOOL class that raised this exception, if the partition raising this exception is interpreted. This value is an empty string for a compiled partition.
string *TOOLMethod*;	TOOL method in which this exception was raised, if the partition raising this exception is interpreted. This value is an empty string for a compiled partition.
i4 *TOOLLine*;	The line of the TOOL method in which this exception was raised, if the partition raising this exception is interpreted. This value is 0 for a compiled partition.
boolean *IsRemote*;	TRUE if the exception was raised on a partition other than the partition acting as an IIOP server. FALSE if the exception was raised on the same partition as the IIOP server.
string *Program*;	Application running the partition that raised the exception, such as ftexec, ftlaunch, or the executable.
string *TaskId*;	Hexadecimal number that identifies the task in which the exception was raised.
string *PartitionId*;	Hexadecimal number that identifies the partition on which the exception was raised.

In general, the members in bold are most useful when you write routines that let your IIOP client programmatically catch and deal with the exception. The rest of the members contain diagnostic information that can help you debug your IIOP server and IIOP client applications.

Running a Client Application

Before you can run an IIOP client with a TOOL server, you need to install and set up the following components:

- iPlanet UDS runtime and TOOL application on a server machine running in an iPlanet UDS environment
- If the server requires it, you need to install and start the IIOP gateway application (iiopgw), if used, on a node in the iPlanet UDS environment (see "Using the iPlanet UDS IIOP Gateway" on page 34)
- ORB product on the client machine
- IIOP client application

➤ **To run an IIOP client with a TOOL application**

1. If the IOR file is generated at runtime, start the iPlanet UDS partition containing the service object. This automatically starts a listener.

   A listener is also automatically started if you start the Gateway application.

   For information on listeners, see "Listener and Service Object Configurations" on page 29.

2. If the server uses the naming service, make sure it is running.
3. Start the client application.

As long as a listener is running, if the IIOP client sends a message to a distributed object in an iPlanet UDS partition that is not running, iPlanet UDS auto-starts the partition.

You can start all the TOOL partitions ahead of time to improve the performance of the interaction between the IIOP client and the TOOL server.

Working with Java Clients

This example describes the steps you need to take to create a Java client that calls a TOOL service object using CORBA.

The IDL File

The following code shows a sample IDL file, hello.idl, which you must compile for the client and the server. The text in this file specifies the interface to the CORBA object(s) you want to invoke; in this case the method is called HelloMethod, it returns a void, and it passes an input string parameter.

```
module HelloModule
{
    interface HelloInterface
    {
      void HelloMethod(in string param);
    };
};
```

Creating a Java Client

The following sections explain how you write a Java client.

Compile the IDL Interface File

Start a command-line shell and run the IDL-to-Java compiler on the idl source file. (For information on flag settings for your compiler, please consult the documentation provided). The following sample commands compile the hello.idl source file.

```
idlj -fclient hello.idl
```

This generates up to six files in a subdirectory that has the same name as the module, in this case, HelloModule. The table below describes the contents of these files:

File name	Contents
`_HelloInterfaceImplBase.java`	An abstract class that functions as the server skeleton, providing basic CORBA functionality for the server. It implements the HelloServer.java interface. This is the abstract class from which the server implementation inherits.
`_HelloInterfaceStub.java`	The client stub that provides CORBA functionality for the client. It implements the HelloServer.java interface.
`HelloInterface.java`	The Java version of the .idl file. It contains the single method, HelloMethod. The Hello.java interface extends org.omg.CORBA.Object, providing standard CORBA object functionality as well.
`HelloInterfaceHelper.java`	A class that provides additional functionality, primarily the narrow() method required to cast CORBA object references to their proper type.
`HelloInterfaceHolder.java`	A class that holds a public instance member of type Hello. It provides operations for out and inout arguments, which CORBA has but which do not map readily to Java's semantics.
`HelloInterfaceOperations.java`	Base class to HelloInterface (JDK 1.3 only).

Implementing the Java Client

The Java client implementation shown below uses the naming service to get an object reference. It initializes the client-side ORB, it gets a reference to the naming service, it obtains an object reference for the object whose methods the client wants to invoke, and it invokes those methods. The subsections that follow discuss the implementation in greater detail and provide code samples that show how you use an IOR file get the object reference.

```
import java.io.*;
import java.util.*;
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

Class HelloClient
{
  public static void main(String[] args)
  {
    try
    {
      //create and initialize the ORB
      String orbArgs[]={"-ORBInitialPort", "1050"};
      org.omg.CORBA.ORB orb =
                  org.omg.CORBA.ORB.init(orbArgs, null);

      //get a reference to the naming context
      org.omg.CORBA.OBJECT objNameService =
              orb.resolve_initial_references("NameService");
      NamingContext ctx =
            NamingContextHelper.narrow(objNameService);

      //Get a reference to the object implementation
      NameComponent nc1 = new NameComponent("ToolHelloServer",
"");
      NameComponent{} name1 = {nc1};
      org.omg.CORBA.Object obj = ctx.resolve(name1);
      HelloModule.HelloInterface svr =
            HelloModule.HelloInterfaceHelper.narrow(obj);

      //call the remote method
      svr.HelloMethod("Hi there, this is Java calling TOOL!");
    }
    catch(org.omg.CORBA.UserException e)
    {
      System.err.println(e);
    }
    catch(org.omg.CORBA.SystemException e)
    {
      System.err.println(e);
      }
    }
};
```

Initializing the ORB

The initialize method creates an ORB instance and uses an args parameter to pass it the host name of a machine running a naming service and the port number on which the initial naming service listens.

The args parameter is defined to be a string type array. This form of the init method is used for standalone applications and may be called from applications only.

```
init(String[] args, Properties props)
```

You must use an alternate form to initialize an ORB that is created for an applet. For more information, see Sun's CORBA API specification.

Getting an Object Reference from the Naming Service

The client code in the previous example uses the naming service to get a reference to a CORBA object. First it gets a reference to the naming service, then it initializes an array of NameComponent objects to specify the name of the object implementation. Finally, the client code passes the array to the resolve method of the naming service. The naming service returns a CorbaObject, which the client must first narrow to the desired type (to obtain a valid reference to the object implementation) and then cast it in order to work with it.

Getting an Object Reference from an IOR File

Using a naming service is the recommended method of obtaining an object reference. However, you might want to use the alternative method of having the server pass the object reference back to the client through an IOR (interoperable object reference) file or by some other means.

The client must use the string_to_object method to turn the string it receives into an object reference. This is shown in the following code sample.

```
// code that converts the IOR string to an object

String IORString = ReadIOR(fileName);
Omg.Omg.CORBA.Object obj = orb.string_to_object(IORString);

SomeClass MySvr = SomeClassHelper.narrow(obj);
```

The code below shows a helper function that the client would use to read the IOR file and obtain the string.

```
// helper method to read a string from an IOR file
public static String ReadIOR(String fileName)
{
    String retValue = null;
    java.io.DataInputStream input = null;

    try
    {
      FileInputStream inFile = new FileInputStream(fileName);
      input = new java.io.DataInputeStream(inFile);
    }
    catch(IOException ex)
    {
      System.err.println(ex);
      return null;
    }

    try
    {
      retVal = input.readLine();
      input.close();
    }
    catch(IOException ex)
    {
      System.err.println(ex);
    }
    return ret Val;
}
```

Starting the Naming Service

If your application uses the naming service to obtain CORBA object references, you need to start the naming service before you test or run your application.

➤ **To start the naming service**

1. Open a command-line shell.
2. Enter the command to start the naming service using the following syntax:

   tnameserv -ORBInitialPort *portnumber*

   For example:

   ```
   tnameserv -ORBInitialPort 1050
   ```

Building the Client

To build the client, compile the .java files, including the stubs and skeletons that were created when you compiled the IDL source file. For example:

```
javac *.java HelloApp/*.java
```

Working With TOOL Clients

This example describes the steps you must complete to create a TOOL client that calls a Java server using CORBA.

The IDL File

The following code shows a sample IDL file, hello.idl, which you must compile for the client and the server. The text in this file specifies the interface to the CORBA object(s) you want to invoke; in this case the method is called HelloMethod, it returns a void, and it passes an input string parameter. It is the responsibility of the server to make the IDL file available to the client.

```
module HelloModule
{
    interface HelloInterface
    {
      void HelloMethod(in string param);
    };
};
```

Creating a TOOL Client

The following subsections explain how you write a TOOL client.

➤ **To compile the IDL interface file and import it into your work repository**

1. Start a command-line shell and invoke the corbagen utility using the following syntax:

   `corbagen -i` *inputfile* `-o` *outputfile*

Taking the file shown in "The IDL File" as an example, you would use the following command:

```
corbagen -i hello.idl -o hello.pex
```

The corbagen utility uses the flags listed in the following table. Normally, you would only use the -i and -o flags.

Flag	Description
`-I` *dir*	Specifies the directory to search for the preprocessor. If your implementation code includes preprocessor statements, you must use this flag.
`-i` *filename*	Specifies the name of the IDL input file.
`-o` [*filename*] [`-o` *filename*]	Specifies the name of the .pex output file that will contain the IDL - TOOL mapping for the interface. If you do not specify a file name, the name *inputfile*.pex will be assigned by default.
`-u`	Prints usage message and exits.

For example, if the IIOP server provides the IDL file hello.idl, shown above, Corbagen parses the file and produces the following TOOL code:

```
begin TOOL HelloModule;
includes Corba;

  forward HelloInterface;
  class HelloInterface inherits from Corba.CorbaObjectImpl
    has public method HelloMethod(
      input param :string
     );

  has property
    distributed=(allow=on, override=on, default=on);

  end class;
  method HelloInterface.HelloMethod(
    input param :string
    )
 begin
 end method;
end ;
```

2. Import the resulting .pex file into your work repository by choosing Plan > Import from the Repository Workshop and specifying the file name.

   Alternately, you can use Fscript to import the generated .pex file. Use the ImportPlan command, as follows:

   ```
   fscript> ImportPlan myFile.pex
   ```

   For more information, see the *Fscript Reference Guide*.

➤ **To create a project for the client and specify the suppliers for your project**

1. Create a project for the client implementation by choosing Plan > New Project from the Repository Workshop and specifying the name of your project.
2. To specify the suppliers to your project, open the Project Workshop and choose File > Supplier Plans. Specify the CosNaming library and then the CORBA library as suppliers.
3. You also need to add the compiled IDL file as a supplier. In the Project Workshop, choose File > Supplier Plans, and specify the name of the .pex file as a supplier.

Implementing the TOOL Client

The TOOL client implementation shown next uses the naming service to get an object reference. It initializes the client-side ORB, it gets a reference to the naming service, it obtains an object reference for the object whose methods it wants to invoke, and it invokes those methods.

The sections that follow discuss the implementation in greater detail and provide code samples that show how you use an IOR file to get the object reference.

```
//initialize client-side ORB
theOrb : Corba.ORB = Corba.ORB();
args : Array of TextData = new;
args.AppendRow (TextData(Value = '-ORBInitialHost
ServerMachine'));
args.AppendRow (TextData(Value = '-ORBInitialPort '));
args.AppendRow (TextData(Value = '1050'))
args.AppendRow (TextData(Value = 'myPassword'))
theOrb.Initialize(args);

//get a reference to a naming service from the ORB
begin
    nameServiceObj : Corba.CorbaObject =
              theOrb.resolve_initial_references('NameService');
    nameServiceObj = theOrb.narrow(nameServiceObj,
NamingContext);
```

```
// typecast the reference returned above
    nameService : CosNaming.NamingContext =
              CosNaming.NamingContext(nameServiceObj);

// get reference to object implementation
    nameComp : NameComponent = new;
    nameComp.id = 'MyServer';
    nameComp.kind = 'text';
    nameArray : Array of CosNaming.NameComponent = new;
    nameArray.AppendRow(nameComp);
    mySvrObj : Corba.CorbaObject =
nameService.resolve(nameArray);
    mySvrObj = theOrb.narrow(mySvrObj,
HelloModule.HelloInterface);

// typecast to the actual interface type
    mySvr : helloModule.HelloInterface =
                  HelloModule.HelloInterface(mySvrObj):

// invoke the method using the object reference
    mySvr.helloMethod('Calling MyServer - do you read me?');
//exception handling
exception when e = NotFound do
//handle exception
exception when e = CannotProceed do
//handle exception
exception when e = InvalidName do
//handle exception
end;
```

Initializing the ORB

The initialize method creates an ORB instance and uses an args parameter to pass it the host name of a machine running a naming service and the port number on which the initial naming service listens.

The args parameter is defined to be an array of TextData values. As shown in the client implementation sample code, you can specify options and values as one array element or use more than one element to express these. For example, note the difference between the number of array elements used to specify the host name and that used to specify the port number.

You can also append additional array elements to specify command-line arguments or other data. The ORB will skip these elements if it does not understand them.

Getting an Object reference from the Naming Service

The client code in our previous example uses the naming service to get a reference to a CORBA object. First it gets a reference to the naming service, then it initializes an array of NameComponent objects to specify the name of the object implementation and its kind, and finally, the client code passes the array to the resolve method of the naming service. The naming service returns a CorbaObject, which the client must narrow and typecast to the desired type to obtain a valid reference to the object implementation.

Note that using ORB.resolve_initial_references() does not currently work with the Visigenic naming service.

Getting an Object Reference from an IOR File

Using a naming service is the recommended method of obtaining an object reference. However, you might want to use the alternative method of having the server pass the object reference back to the client through an IOR (interoperable object reference) file. The code below shows how the client would read the file and convert the string passed in the file into an object reference:

```
// use an IOR file to get object reference
f: File = new;
iorString : TextData = new;
f.SetPortableName(portableFileName);
f.Open;
f.ReadLine(iorString, FALSE);
f.Close

//use ORB method to convert the string to an object
theOrb : ORB = new;
obj: CorbaObject = theOrb.string_to_object(iorString.Value);

//narrow the object to the right type
obj = theOrb.narrow(obj, HelloModule.HelloInterface);

//Typecast the object to the right type
svr: HelloModule.HelloInterface =
HelloModule.HelloInterface(obj);

//Call method
svr.HelloMethod('Hi there, this is TOOL!');
```

See "Using an IOR File to Pass an Object Reference" on page 58 for the server-side code required to pass an object reference as a string.

Starting the Naming Service

If your application uses the naming service to obtain CORBA object references, you need to start the naming service before you test or run your application.

➤ **To start the naming service of the Sun SDK (1.3 and 1.2.2)**

1. Open a command-line shell.
2. Enter the command to start the naming service using the following syntax:

   tnameserv -ORBInitialPort *portnumber*

   For example:

   ```
   tnameserv -ORBInitialPort 1050
   ```

Testing, Making a Distribution, and Installing the TOOL Client

The steps for testing, making an application distribution, and installing the iPlanet UDS IIOP client application are the same as for any other iPlanet UDS application, except as described in this section.

Testing Your Application

Before you can test your application, you must start the server application.

- If you are using an IOR file to get an object reference, you must get its location from the server provider.
- If you are using the naming service, you must make sure it is running.

In the Partition Workshop, you can test run your client application using the Run > Run command.

In Fscript, you can test run your application using the `RunDistrib` command, as described in the *Fscript Reference Guide*.

You should also deploy your client application and test it by making a distribution and installing it, as described in the next section.

Making a Distribution and Installing Your Application

Using auto-install If you are installing your application to test it in your active environment, you can do so using the auto-install feature.

In the Partition Workshop, you can use the File > Make Distribution command, then enable the Install in Current Environment toggle and the Make button to automatically install the application after you make the distribution. See *A Guide to the iPlanet UDS Workshops* for more information about making a distribution.

In Fscript, you can use the `MakeAppDistrib` command, specifying the auto-install flag to automatically install the application after you make the distribution, as follows:

```
fscript> MakeAppDistrib 0 "" 0 1
```

See the *Fscript Reference Guide* for more information about using Fscript and the `MakeAppDistrib` command.

Without auto-install If you do not want to install the application now, you can just make a distribution, then install the application later in the environment where it is to run. To do so, you can make a distribution without using the auto-install option, then use Escript or the Environment Console to load and install the application into your environment.

NOTE If you chose to create the IOR file at `distribution`, your deployment and development environments must be the same environment. For information on choosing when to create the IOR file, see the description of the Create At field under "Using the IIOP Configuration Dialog" on page 48.

If you specified `runtime` as an IOR File configuration setting, iPlanet UDS generates the IOR string when you start the partition containing the service object. If you are deploying the application, runtime is usually the best choice.

For information about installing applications using Escript, see the *Escript and System Agent Reference Guide*. For information about installing applications using the Environment Console, see the *iPlanet UDS System Management Guide*.

Appendix A

IDL and TOOL

This appendix describes the mapping that is done between IDL types and TOOL types, and between TOOL types and IDL types.

IDL to TOOL Mappings

The corbagen tool, which takes an IDL text file as input and generates the required stub, skeleton, and other helper classes, supports the following mapping of IDL to TOOL. These mappings are described in Table A-1.

Table A-1 IDL to TOOL Mapping Enhancements

IDL Type	TOOL Type	Discussion
long	i4	
unsigned long	ui4	
float	float	
double	double	
short	i2	
unsigned short	ui2	
char	ui1	
boolean	boolean	
octet	ui1	
any	pointer	
Object	Corba.CorbaObject	

Table A-1 IDL to TOOL Mapping Enhancements *(Continued)*

IDL Type	TOOL Type	Discussion
struct	CorbaFlat class	For more information, see "CorbaFlat Objects" on page 89 and "Structs" on page 91.
sequence string	array of TextData	
sequence [unsigned] long	array of IntegerData	
sequence [unsigned] short	array of ShortData	The ShortData class is a new TOOL class. It is a subclass of IntegerData, and differs from this class only in that it is serialized as a 2-byte integer. The Value field in ShortData is still 4 bytes. This means that the value field can store 4 bytes, but it can only read/write 2 bytes. It is up to the user to make sure that the value read or written does not exceed the 2 byte range.
sequence char	BinaryData	Each element of the sequence char/octet/boolean type is mapped to BinaryData byte. This mapping is more efficient and incurs less overhead than mapping these sequences into TOOL arrays. In addition, the BinaryData class provides a wealth of methods for data manipulation.
sequence octet	BinaryData	
sequence boolean	array of BooleanData	
sequence double	array of DoubleData	
sequence float	array of FloatData	The FloatData class is a new TOOL class. It is a subclass of DoubleData, and differs from this class only in that it is serialized as a 4-byte float. The Value field in FloatData is still 8 bytes. This means that the value field can store 8 bytes, but it can only read/write 4 bytes. It is up to the user to make sure that the value read or written does not exceed the 4 byte range.
sequence *other*	array of Object	

Working with Sequences

Both bounded and unbounded sequences are mapped to the types indicated in Table A-2. It is up to you to make sure that input parameters that are bounded sequences have the correct length.

Table A-2 Mapping of Sequences

IDL Type	TOOL Type	Discussion
sequence string	array of TextData	
sequence [unsigned] long	array of IntegerData	
sequence [unsigned] short	array of ShortData	The ShortData class is a new TOOL class. It is a subclass of IntegerData, and differs from this class only in that it is serialized as a 2-byte integer. The Value field in ShortData is still 4 bytes. This means that the value field can store 4 bytes, but it can only read/write 2 bytes. It is up to the user to make sure that the value read or written does not exceed the 2 byte range.
sequence char	BinaryData	Each element of the sequence char/octet/boolean type is mapped to BinaryData byte. This mapping is more efficient and incurs less overhead than mapping these sequences into TOOL arrays. In addition, the BinaryData class provides a wealth of methods for data manipulation.
sequence octet	BinaryData	
sequence boolean	array of BooleanData	
sequence double	array of DoubleData	
sequence float	array of FloatData	The FloatData class is a new TOOL class. It is a subclass of DoubleData, and differs from this class only in that it is serialized as a 4-byte float. The Value field in FloatData is still 8 bytes. This means that the value field can store 8 bytes, but it can only read/write 4 bytes. It is up to the user to make sure that the value read or written does not exceed the 4 byte range.
sequence *other*	array of Object	

Please note that both bounded and unbounded sequences are mapped to the types indicated in Table A-1. It is up to you to make sure that input parameters that are bounded sequences have the correct length.

Working With Unions

An IDL union is translated into a TOOL class of the same name. All classes that have been created from an IDL union share the following method and virtual attributes:

Name	Kind	Description
clear_union	method	After this call, the value of the union is cleared and the union is no longer initialized.
discriminator	virtual attribute	A read-only field that specifies which case of the union is set. The return value depends on the switch case specified in the IDL declaration.
is_initialized	virtual attribute	TRUE if the value of the union has been set; FALSE if it has not been set or if the clear_union method has been called.
is_default	virtual attribute	TRUE if the discriminator is not set to one of the explicit cases listed in the IDL declaration. With reference to the example below, the is_default attribute would return TRUE if the discriminator is set to any value other than 1 or 2.

Here is an example of an IDL union.

```
union my_union switch(long)
{
    case1:
            short short_value;
    case2:
            long long_value;
    case3:
            float def_value;
};
```

This union is translated into a TOOL class with the name my_union that has the following virtual attributes and methods:

- Virtual attributes: short_value, long_value, and def_value.

  Setting any of these virtual attributes, automatically sets the discriminator to the corresponding value. If you try to get any of the attributes when the union is not initialized or when it is set to a different case, will result in a Corba.BAD_OPERATION exception. Passing an uninitialized union to a CORBA call will also result in an exception.

- Methods: set_def_value

  This is the same as the def_value attribute, except that it allows you to set the discriminator to a given value, whereas using the def_value attribute picks an arbitrary value. Passing a discriminator that corresponds to any of the explicitly stated cases will result in a Corba.BAD_OPERATION exception.

TOOL to IDL Mapping

The following sections describe how parameter, return values, and exceptions are translated into IDL.

Parameter and Return Values

Corbagen generates IDL only for methods that have parameters and return values of the following supported classes and data types (described in the sections that follow):

- portable scalar values (For more information, see "Portable Scalar Values" below)
- DataValue subclasses (page 87)
- the iPlanet UDS classes Object and GenericException and its subclasses (page 88)
- distributed objects (page 89)
- CorbaFlat objects (page 89)
- Structs (page 91)
- Arrays (page 91)

There are additional restrictions on generating IDL:

- iPlanet UDS does not generate IDL for overloaded methods.
- You can pass structs from clients to iPlanet UDS distributed objects, and the reverse. If the client passes a struct as an input parameter to a method of an iPlanet UDS distributed object, that struct exists only for the duration of the method call. At the end of the method call, iPlanet UDS deallocates the memory for the struct in the iPlanet UDS partition.

  If you want to access the struct after the method completes execution, the method must explicitly allocate memory for the data and copy the struct into that memory.

Writing wrapper methods If a method has parameters and return values that do not conform to these types and guidelines, no IDL is generated for that method. However, you can write a wrapper method for it.

For example, if a class provides overloaded methods, you can write a wrapper method that defines a unique method signature for each of the overloaded methods. iPlanet UDS can then generate IDL for these wrapper methods.

Portable Scalar Values

iPlanet UDS performs the following mapping between TOOL portable scalar data types and IDL:

TOOL Scalar Data Type	IDL Equivalent
boolean	boolean
double	double
float	float
i2	short
i4	long
integer	long
string	string
ui2	unsigned short
ui4	unsigned long

NOTE iPlanet UDS does not produce IDL for methods that have parameters or return values with data types that are not portable across different platforms, such as the TOOL data types int, long, or short.

iPlanet UDS Framework Library DataValue Subclasses

The following table lists the DataValue subclasses that are supported and the IDL that is generated for each:

Supported iPlanet UDS DataValue Subclass	**IDL Equivalent**
BinaryData	sequence<octet>
BinaryNullable	struct BinaryNullable_struc{ boolean nullind; BinaryData value; (Note: resolves to a sequence<octet>.) }
BooleanData	boolean
BooleanNullable	struct BooleanNullable_struc{ boolean nullind; boolean value; }
DateTimeData	string
DateTimeNullable	struct DateTimeNullable_struc{ boolean nullind; string value; }
DecimalData	string
DecimalNullable	struct DecimalNullable_struc{ boolean nullind; string value; }
DoubleData	double
DoubleNullable	struct DoubleNullable_struc{ boolean nullind; double value; }

Supported iPlanet UDS DataValue Subclass	IDL Equivalent
IntegerData	long

IntegerNullable	struct IntegerNullable_struc{
	    boolean nullind;
	    long value;
	}

TextData	string

TextNullable	struct TextNullable_struc{
	    boolean nullind;
	    string value;
	}

The ImageData, ImageNullable, and IntervalData subclasses of DataValue are not supported.

Objects of the DataValue subclasses shown in the table above are automatically converted to the scalar data types or structs when they are passed to the client, and converted from these structs or scalars to an object with the correct data type when they are received from clients.

If you pass a NIL for parameters that are DataValue subclasses, the client receives a struct of binary zeros.

The Object Class and GenericException and Its Subclasses

iPlanet UDS generates special IDL for the following classes:

TOOL Class	IDL Equivalent
Object	ForteObject
Any subclass of GenericException	GenericException

For more information about how iPlanet UDS maps exceptions to the generated GenericException class, see "Working with Exceptions" on page 92.

Distributed Objects

Methods returning references to unanchored objects or returning copies of distributed objects (except for the DataValue subclasses described previously on page 87), do not have corresponding IDL generated for them.

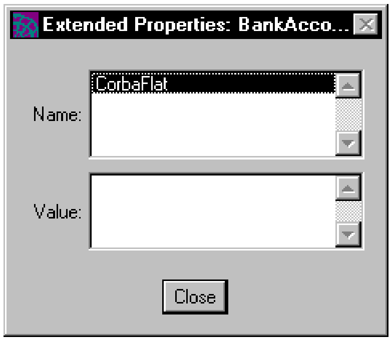
CorbaFlat Objects

You can send or receive an iPlanet UDS non-distributed object as a parameter. This object is converted to a CORBA struct, or "flattened" in the IDL file. This struct includes all the attributes defined in the object's class.

To enable a nondistributed object to be translated into a struct and sent as a parameter, you must specify the CorbaFlat extended property in the class definition. If you use the CorbaFlat extended property, the distributed property must *not* be specified as *allowed* or *default*.

```
class MyClass inherits Object
  . . .
  -- methods and attributes here
  . . .
  has property extended=(CorbaFlat);
```

You can also specify the CorbaFlat extended property in the iPlanet UDS Workshops in the Extended Properties dialog. In the Project Workshop select the class, then choose the Component > Extended Properties command.


Extended Properties: BankAcco...

Name: CorbaFlat

Value:

For example, the BankAccount class has the following class statement:

```
begin CLASS;
class BankAccount inherits from Framework.Object
has public  attribute AcctBalance: Framework.double;
has public  attribute AcctName: Framework.TextData;
has public  attribute AcctNumber: Framework.integer;
has public  method Init;
has property
  shared=(allow=off, override=on);
  transactional=(allow=off, override=on);
  monitored=(allow=off, override=on);
  distributed=(allow=off, override=on);
  extended = (CorbaFlat);
end class;
```

The IDL generated for this flattened class is as follows:

```
typedef struct BankAccount_struct
  {
    double AcctBalance;
    Framework::TextData AcctName;
    integer AcctNumber;
  } BankAccount;
```

A class that is to be translated into a CORBA struct (flattened) cannot have any attributes whose type is also that class. For example, the following class definition cannot be flattened because the attribute Brother is of the type Child, which is the class itself:

```
class Child inherits Object
  Brother : Child; -- The type is the class Child.
  has property extended=(CorbaFlat);
end class;
```

Similarly, a class that is to be flattened cannot have any attributes of a type that is a class that has any attributes whose type is the class to be flattened. In the following example, the class definitions cannot be flattened because the attribute Parent in class Child is of the type Adult, while the class Adult also has an attribute Offspring that is of type Child:

```
forward Adult;
class Child inherits Object
  Parent : Adult; -- The type is the class Adult.
  has property extended=(CorbaFlat);
end class;

class Adult inherits Object
  Offspring : Child; -- The type is the class Child;
  has property extended=(CorbaFlat);
end class;
```

In either of these cases, iPlanet UDS generates the exception DistributedAccessException at runtime.

Structs

iPlanet UDS generates CORBA structs that represent iPlanet UDS structs (described in *Integrating with External Systems*).

Arrays

iPlanet UDS automatically generates IDL for methods that use parameters with an Array data type and for attributes that have an Array data type that is a distributed object reference (page 89) or a flattened object (page 89), or that is one of the supported DataValue subclasses (page 87). This generated IDL is a sequence rather than an IDL Array.

Parameters and attributes that have an Array data type must also refer to the class type of the objects in the array. For example, an attribute with the data type Array of TextData will be included in the IDL output as a sequence. However, an attribute with the data type Array will not be included.

The following example shows the TOOL code for an attribute in the BankServices class that has an Array data type:

```
has public attribute AcctList: Array of BankServices.BankAccount;
```

The following example shows how the BankServices.AcctList attribute is expressed in the generated IDL file:

```
attribute BankServices::sequence_BankAccount AcctList;
```

Working with Exceptions

When iPlanet UDS generates IDL for a method that raises an exception, the IDL equivalent of the method signature by default raises one of two IIOP exceptions: either GenericException or UserException.

You can specify which throws clause will be exported to IDL by setting any of the following three extended properties (for a description of how to set an extended property on a class, see "CorbaFlat Objects" on page 89):

DefaultThrowsClause This property is set at the project level and controls the default throws clause in the IDL output (determines which exceptions are thrown) for every method in the project. By default the clause specifies GenericException and UsageException, as follows:

```
(Framework::GenericException, Framework::UsageException)
```

ThrowsClause This clause is set at a method level and overrides the default clause or, if it is set, the DefaultThrowsClause.

IsThrowable This property is set on a class that you want to export to IDL as an exception. The class must not be distributed.

An iPlanet UDS exception class when translated into IDL contains an ErrorDesc_struct that is defined in the framewor.idl file. This ErrorDesc_struct contains the data in the attributes of the Framework ErrorDesc class. An client must be able to respond to exceptions when they are raised by an iPlanet UDS method. Ensure that the exceptions raised by an iPlanet UDS method define appropriate information in the following attributes of the raised TOOL exception:

Attribute	Description
Message	Message text for the exception.
ReasonCode	Code indicating the cause of the exception.
Severity	Code indicating the severity of the exception.
SetNumber	Number indicating the message set containing the message associated with this exception.
MsgNumber	Number of the message in the message set associated with this exception.

For a description of how an client interprets this information, see "Interpreting Exception Information" on page 65.

For more information about raising exceptions, see the *TOOL Reference Guide* and the Framework Library online Help.

Appendix B

Using Fscript to Configure CORBA Servers

This appendix contains detailed information about how you use Fscript commands to specify the settings for service objects that are acting as CORBA servers.

Configuration Parameters for a TOOL CORBA Object

This section describes how to use the configuration parameters of the Fscript `SetServiceEOSInfo` command to set up iPlanet UDS service objects to act as CORBA objects.

You can specify the following configuration parameters in the IIOP Configuration dialog or in the *export_name* argument of the Fscript `SetServiceEOSInfo` command for service objects that are CORBA objects:

- IORFile
- ListenLocation (Location field in the Service Object Properties dialog)
- Forward (Turn off the Redirect Request toggle in the Service Object Properties dialog)
- Redirect (Turn on the Redirect Requests toggle in the Service Object Properties dialog)
- Host
- Port

- DisableAutoStartGW (Turn on the Disable Auto Start toggle in the Service Object Properties dialog)
- IIOP mode

The default configuration parameters for an iPlanet UDS service object that is a CORBA object are: "`forward`, `listenlocation = here, inbound, generate=idl`"

The syntax for the `SetServiceEOSInfo` Fscript command for an iPlanet UDS service object that is a CORBA object is the following:

```
SetServiceEOSInfo service_object_name iiop
   "[outbound,] IORFile = (name=iorfile_name [, runtime | dist])
   [, listenlocation = here | remote] [, forward | redirect]
   [, host = host_name] [, port = port_number]
   [, disableautostartgw]"
```

service_object_name is the name of the service object that you want to make available to CORBA clients. If the current project contains the service object, you can specify just the name of the service object; otherwise, *service_object_name* should specify the project name and the service object name, separated by a period.

The details of the configuration parameters specified in the quotation marks are described in the following sections.

Outbound Parameter

This parameter has been deprecated: please do not use. Existing applications that use it will still work.

By default, iPlanet UDS makes the service object an IIOP server. If you want the service object to become a CORBA client that sends requests to (does callout to) a CORBA server, specify this parameter.

IORFile Parameter

The `IORFile` parameter specifies that the runtime system write an IOR (Internet Object Reference) string in a file for this service object. This parameter specifies when the file is written and the name of the file.

A CORBA client uses the information in the IOR string to determine the location of a listener (using the specified host and port). The client then sends requests to that listener, which in turn routes the requests to the partition containing the service object.

NOTE If you have multiple service objects generating IOR files, you must specify a unique path and file name for the `name` parameter of the `IORFile` parameter; otherwise, the IOR files will overwrite each other.

You can specify this configuration parameter with the `SetServiceEOSInfo` command or in the IIOP Configuration dialog:

IORFile = (runtime | distribution, name=*iorfile_name*)

Parameter name or keyword	Description
`runtime`	Generates this file when the partition containing the service object starts. `runtime` and `distribution` are mutually exclusive keywords.
`distribution`	Generates this file when the application distribution for the partition containing the service object is generated. `runtime` and `distribution` are mutually exclusive keywords.
`name=`*iorfile_name*	Specifies the name of the IOR file in local format. If the file name is relative, the file is created in the $FORTE_ROOT/etc/iiopior directory. If the file name is absolute, the file is created with the specified local file name.

If you specify only the `name` parameter or only the `runtime` or `distribution` option for IOR file keywords, you do not need to use parentheses around the `IORFile` parameter value. For example, you can specify IORFile=runtime, if you also specify IORFile=name=myfile.ior on the same command, as shown. Note that the command is invoked on one line.

```
fscript> SetServiceEOSInfo myServer iiop “Forward, IORFile=runtime,
   ListenLocation=here, IORFile=name=myserv.ior”
```

ListenLocation Parameter

The `listenlocation` parameter specifies whether the CORBA client uses a listener on the same partition as the service object, or a listener on a remote partition or gateway.

You can specify this configuration parameter in one of the following ways, as part of the `SetServiceEOSInfo` command's *export_name* value or in the IIOP Configuration dialog:

```
listenlocation = here | remote
```

Parameter name or keyword	Description
`here`	CORBA clients use a listener on the same partition. `here` and `remote` are mutually exclusive. `here` is the default.
`remote`	CORBA clients use a listener on remote partition or an IIOP gateway. `here` and `remote` are mutually exclusive.

IIOP gateway If you want to have a listener run in a separate partition from your application, you can set up an application referred to as the IIOP gateway. The IIOP gateway is an application provided by iPlanet UDS that is named iiopgw. The application starts a listener.

When you specify the `remote` keyword for the `listenlocation` parameter, the CORBA client can use a listener belonging to the IIOP gateway application on the specified host. If no listener is running at the specified port on the specified host and the IIOP gateway is installed on that machine, iPlanet UDS automatically starts the IIOP gateway. You can specify the `DisableAutoStartGW` keyword (described in "Disableautostartgw Parameter" on page 101) to prevent iPlanet UDS from automatically starting the IIOP gateway this way.

The most common reason to use the IIOP gateway application is to provide a single external listener for all CORBA clients. You can install an IIOP gateway application on a particular machine and have service objects specify the `listenlocation=remote` parameter and the host name and port number of the listener started by the IIOP gateway.

Forward Parameter

The `forward` keyword specifies that the listener route each request to the partition containing the appropriate distributed object. The listener routes the requests to other iPlanet UDS partitions, as necessary. The `forward` and `redirect` keywords are mutually exclusive, and `forward` is the default

You can specify `forward` or `redirect` to improve the performance of the client interaction with the iPlanet UDS application and to work with Java security.

NOTE If your IIOP client application frequently references TOOL distributed objects that are on different partitions than the listener or listeners that it uses, you can specify the `redirect` keyword to reduce the amount of time required for a request to reach the intended iPlanet UDS object. After the first request, the client sends all subsequent requests directly to a listener on the correct partition instead of sending them to the original listener.

If the listener for the iPlanet UDS application is on a Web server that uses Java security, Java security prevents the IIOP client from directly accessing partitions running on other machines. You should consider the following options when deciding how to set the routing mode:

- If you specify the `redirect` keyword, all the partitions that you want IIOP clients to access must be placed on the same machine as the Web server to ensure that the IIOP clients can access all the partitions they need to.
- If you specify the `forward` keyword, the partitions that you want IIOP clients to access can be on different machines from the Web server. You can have IIOP clients access a single listener that resides on the Web server machine, perhaps a listener started by an IIOP gateway. iPlanet UDS then routes the requests to the other iPlanet UDS partitions.

Redirect Parameter

The `redirect` parameter specifies that the CORBA client be informed of the location of the distributed object, after which the client sends requests directly to a listener on the same partition as the distributed object instead of routing all requests though the original listener. The `forward` and `redirect` parameters are mutually exclusive.

The listener tells the CORBA client to send the request again to the listener on the same partition as the appropriate iPlanet UDS object. If there is no listener running on that partition, this option automatically starts a listener for that object. The client then sends all subsequent requests for this iPlanet UDS object directly to the partition containing the object.

The CORBA client cannot failover to other service objects in this mode.

Host Parameter

The `host` parameter specifies the TCP name (or the IP address) of the host machine on which the listener is running. This host name is included in the IOR file, so the IIOP client will send requests to that host.

You can only specify a host name if the listener location is Remote.

If the IOR file is to be created at distribution time, and you do not specify a host name, then the name of the host on which the application distribution is made is put into the IOR file.

If the IOR is to be created at runtime, and you do not specify a host name, the name of the host on which the partition containing the service object is running is placed in the IOR file.

Port Parameter

The port parameter specifies the port number for a listener to which the CORBA client can send requests. The port location is included in the generated IOR file, so the client will send requests to that port number.

You must specify a port number when the IOR file is created at distribution time or if you have specified a remote listener.

If you do not specify a port number when the listener is here and the IOR file is generated at runtime, the operating system assigns a port number at runtime, and iPlanet UDS includes that port number in the IOR file.

Disableautostartgw Parameter

The `disableautostartgw` keyword prevents iPlanet UDS from automatically trying to start a gateway. You can only use this keyword with a remote listener (`listenlocation=remote`). If you do not specify this keyword, iPlanet UDS automatically tries to start the IIOP gateway application if a listener is not running where a service object expects one. If the IIOP gateway application is not installed on a node where iPlanet UDS tries to start it, iPlanet UDS raises a DistributedAccessException.

Configuration Parameters for IIOP Client Service Objects

This section describes configuration parameters you can set for TOOL service objects that represent and are bound to objects provided by external CORBA servers.

In Fscript, you can set the properties for the service object so that it binds to the correct CORBA server by using the `SetServiceEOSInfo` command with the following syntax:

```
SetServiceEOSInfo service_object_name iiop "outbound,
iorfile=(name=iorfile_name)
[, generate = idl]"
```

Argument or Parameter values	**Description**
service_object_name	The name of the service object that you want to bind to the CORBA server described in the IOR file.
iorfile_name	The path and name of the IOR file in local format.

Outbound Parameter

The Outbound keyword specifies that this service object represents an object that is provided by the IIOP server application.

IORFile Parameter

The `IORFile` parameter specifies that iPlanet UDS locate and read the IOR (Internet Object Reference) string in the IOR file at the specified location. The iPlanet UDS IIOP client uses the information in the IOR string to locate and bind to the appropriate IIOP server object.

The documentation for the IIOP server application should provide information about the location and name of the IOR file.

You can specify this configuration parameter as part of the `SetServiceEOSInfo` command's *export_name* value or in the IIOP Configuration dialog:

`IORFile = name=`*iorfile_name*

Parameter name or keyword	Description
iorfile_name	Specifies the name of the IOR file in local format. If the file name is relative, the file is assumed to reside in the $FORTE_ROOT/etc/iiopior directory. If the file name is absolute, the IOR string is read from the file at the specified location that has the specified name.

Generate Parameter

The `generate` parameter specifies the IIOP mode. IDL is the default (and only) IIOP mode.

Index