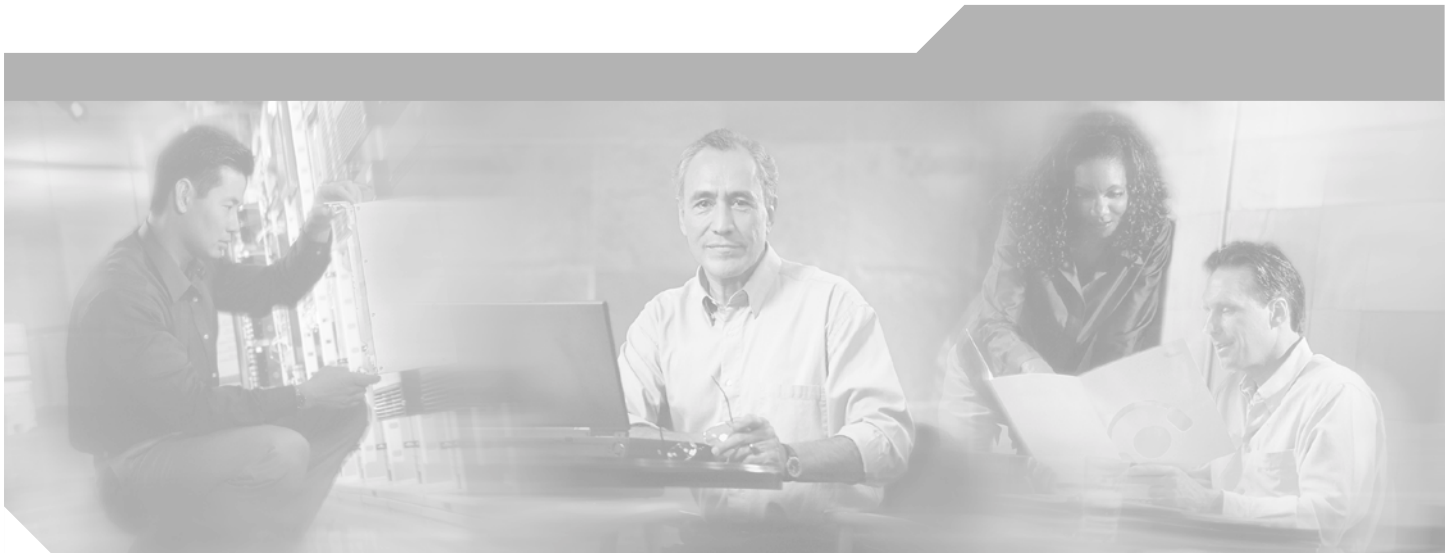


Cisco Element Management Framework CORBA Gateway Developer Guide Version 2.1

Corporate Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 526-4100

Customer Order Number:
Text Part Number: OL-4321-01

Cisco Element Management Framework CORBA Gateway Developer Guide

About This Guide

Cisco Element Management Framework (Cisco EMF) is a highly flexible and extensible platform upon which to build applications, for element management or for higher level network or service management. The CORBA Gateway provides a means for applications to access the Cisco EMF Metadata Service, the Data Abstractor, the Action Launcher, the Deployment Framework, the Participation Service and Object Groups. This Guide provides information for the developer to develop applications for the CORBA Gateway.

Who Should Use This Book

This book is written as a technical resource for developers.

Organization and Use

This guide is organized as follows:

Table 1 Document Organization

Chapter Number	Chapter Title	Content
Chapter 1	Introduction	Discusses the requirements for Cisco EMF CORBA Gateway
Chapter 2	Basic Concepts	Provides an overview of concepts in Cisco EMF which should be understood prior to starting development of a Cisco EMF-based integration solution using the CORBA Gateway
Chapter 3	Tutorials: Getting Started	Details the steps to build a simple CORBA client using C++ or Java
Chapter 4	Accessing the Cisco EMF Servers	Details the CorbaGatewayManager API
Chapter 5	Creating Cisco EMF Objects	Discusses the Participation Framework and Deployment Model in more detail
Chapter 6	Naming and Identifying Cisco EMF Objects	Provides information about the AtlNaming API

Table 1 Document Organization (continued)

Chapter Number	Chapter Title	Content
Chapter 7	Accessing and Modifying Attributes of Cisco EMF Objects	Provides information about the DataAbstractor API
Chapter 8	Launching Cisco EMF Element Manager Actions	Provides information about the ActionLauncher API
Chapter 9	Event Channels	Provides information about the Cisco EMF CORBA Event Channel Service
Chapter 10	Object Groups	Provides information about the Object Groups API
Chapter 11	Example: Provisioning Client Written in C++	Provides a real-life example
Chapter 12	Troubleshooting and Helpful Hints	Describes possible problems and remedies
Appendix A	Other Resources	Lists recommended reading and WWW resources

Conventions

Conventions are presented in the following sections:

- Command Conventions
- Example Conventions
- Document Conventions

Command Conventions

Commands use these conventions:

Table 2 Command Conventions

Format	Description	Example
Boldface font	Commands, keywords, and user entries in text	**/usr/bin**
Italic font	Arguments for which users supply values	*CEMF_ROOT*
Square brackets ([ ])	Optional keywords or arguments	[?]
Braces ({ })	Alternative but required keywords	{yes | no}
Vertical bar (|)	Separator between alternative but required keywords	{yes | no}
Angle brackets (<>)	Non-printing user entries (such as passwords)	<rootpassword>

Example Conventions

Examples use these conventions:

Table 3 Example Conventions

Format	Description	Example
`Plain screen` font	Onscreen displays, examples, and scripts	`CGM Manager`
`Bold screen` font	User entries in examples and scripts	**./cemf install**
Square brackets ([ ])	Default responses	`[tftp idle]`

Document Conventions

This guide uses these conventions:

Table 4 Document Conventions

Format	Description	Example
Boldface font	Menu options, button names, and names of keys on keyboards	**Exit**
Italic font	Directories, filenames, and titles	*Cisco Element Manager System User Guide*

Notes and cautionary statements use these conventions:

Note Means reader take note. Notes contain helpful suggestions or references to materials not contained in this manual.

Caution Means reader be careful. You are capable of doing something that might result in equipment damage or loss of data.

The Cisco TAC Escalation Center addresses priority level 1 or priority level 2 issues. These classifications are assigned when severe network degradation significantly impacts business operations. When you contact the TAC Escalation Center with a P1 or P2 problem, a Cisco TAC engineer automatically opens a case.

To obtain a directory of toll-free Cisco TAC telephone numbers for your country, go to this URL:

http://www.cisco.com/warp/public/687/Directory/DirTAC.shtml

Before calling, please check with your network operations center to determine the level of Cisco support services to which your company is entitled: for example, SMARTnet, SMARTnet Onsite, or Network Supported Accounts (NSA). When you call the center, please have available your service agreement number and your product serial number.

Obtaining Documentation

Cisco provides several ways to obtain documentation, technical assistance, and other technical resources. These sections explain how to obtain technical information from Cisco Systems.

Cisco.com

You can access the most current Cisco documentation on the World Wide Web at this URL:

http://www.cisco.com/univercd/home/home.htm

You can access the Cisco website at this URL:

http://www.cisco.com

International Cisco websites can be accessed from this URL:

http://www.cisco.com/public/countries_languages.shtml

Documentation CD-ROM

Cisco documentation and additional literature are available in a Cisco Documentation CD-ROM package, which may have shipped with your product. The Documentation CD-ROM is updated regularly and may be more current than printed documentation. The CD-ROM package is available as a single unit or through an annual or quarterly subscription.

Registered Cisco.com users can order a single Documentation CD-ROM (product number DOC-CONDOCCD=) through the Cisco Ordering tool:

http://www.cisco.com/en/US/partner/ordering/ordering_place_order_ordering_tool_launch.html

All users can order monthly or quarterly subscriptions through the online Subscription Store:

http://www.cisco.com/go/subscription

Ordering Documentation

You can find instructions for ordering documentation at this URL:

http://www.cisco.com/univercd/cc/td/doc/es_inpck/pdi.htm

You can order Cisco documentation in these ways:

- Registered Cisco.com users (Cisco direct customers) can order Cisco product documentation from the Networking Products MarketPlace:

  http://www.cisco.com/en/US/partner/ordering/index.shtml

- Nonregistered Cisco.com users can order documentation through a local account representative by calling Cisco Systems Corporate Headquarters (California, U.S.A.) at 408 526-7208 or, elsewhere in North America, by calling 800 553-NETS (6387).

Documentation Feedback

You can submit comments electronically on Cisco.com. On the Cisco Documentation home page, click **Feedback** at the top of the page.

You can e-mail your comments to bug-doc@cisco.com.

You can submit comments by using the response card (if present) behind the front cover of your document or by writing to the following address:

Cisco Systems
Attn: Customer Document Ordering
170 West Tasman Drive
San Jose, CA 95134-9883

We appreciate your comments.

Obtaining Technical Assistance

Cisco provides Cisco.com, which includes the Cisco Technical Assistance Center (TAC) website, as a starting point for all technical assistance. Customers and partners can obtain online documentation, troubleshooting tips, and sample configurations from the Cisco TAC website. Cisco.com registered users have complete access to the technical support resources on the Cisco TAC website, including TAC tools and utilities.

Cisco.com

Cisco.com offers a suite of interactive, networked services that let you access Cisco information, networking solutions, services, programs, and resources at any time, from anywhere in the world.

Cisco.com provides a broad range of features and services to help you with these tasks:

- Streamline business processes and improve productivity
- Resolve technical issues with online support
- Download and test software packages
- Order Cisco learning materials and merchandise
- Register for online skill assessment, training, and certification programs

To obtain customized information and service, you can self-register on Cisco.com at this URL:

http://tools.cisco.com/RPF/register/register.do

Technical Assistance Center

The Cisco TAC is available to all customers who need technical assistance with a Cisco product, technology, or solution. Two types of support are available: the Cisco TAC website and the Cisco TAC Escalation Center. The type of support that you choose depends on the priority of the problem and the conditions stated in service contracts, when applicable.

We categorize Cisco TAC inquiries according to urgency:

- Priority level 4 (P4)—You need information or assistance concerning Cisco product capabilities, product installation, or basic product configuration. There is little or no impact to your business operations.
- Priority level 3 (P3)—Operational performance of the network is impaired, but most business operations remain functional. You and Cisco are willing to commit resources during normal business hours to restore service to satisfactory levels.

- Priority level 2 (P2)—Operation of an existing network is severely degraded, or significant aspects of your business operations are negatively impacted by inadequate performance of Cisco products. You and Cisco will commit full-time resources during normal business hours to resolve the situation.
- Priority level 1 (P1)—An existing network is "down," or there is a critical impact to your business operations. You and Cisco will commit all necessary resources around the clock to resolve the situation.

Cisco TAC Website

The Cisco TAC website provides online documents and tools to help troubleshoot and resolve technical issues with Cisco products and technologies. To access the Cisco TAC website, go to this URL:

http://www.cisco.com/tac

All customers, partners, and resellers who have a valid Cisco service contract have complete access to the technical support resources on the Cisco TAC website. Some services on the Cisco TAC website require a Cisco.com login ID and password. If you have a valid service contract but do not have a login ID or password, go to this URL to register:

http://tools.cisco.com/RPF/register/register.do

If you are a Cisco.com registered user, and you cannot resolve your technical issues by using the Cisco TAC website, you can open a case online at this URL:

http://www.cisco.com/tac/caseopen

If you have Internet access, we recommend that you open P3 and P4 cases online so that you can fully describe the situation and attach any necessary files.

Cisco TAC Escalation Center

The Cisco TAC Escalation Center addresses priority level 1 or priority level 2 issues. These classifications are assigned when severe network degradation significantly impacts business operations. When you contact the TAC Escalation Center with a P1 or P2 problem, a Cisco TAC engineer automatically opens a case.

To obtain a directory of toll-free Cisco TAC telephone numbers for your country, go to this URL:

http://www.cisco.com/warp/public/687/Directory/DirTAC.shtml

Before calling, please check with your network operations center to determine the Cisco support services to which your company is entitled: for example, SMARTnet, SMARTnet Onsite, or Network Supported Accounts (NSA). When you call the center, please have available your service agreement number and your product serial number.

Obtaining Additional Publications and Information

Information about Cisco products, technologies, and network solutions is available from various online and printed sources.

- The *Cisco Product Catalog* describes the networking products offered by Cisco Systems, as well as ordering and customer support services. Access the *Cisco Product Catalog* at this URL:

  http://www.cisco.com/en/US/products/products_catalog_links_launch.html

- Cisco Press publishes a wide range of networking publications. Cisco suggests these titles for new and experienced users: *Internetworking Terms and Acronyms Dictionary, Internetworking Technology Handbook, Internetworking Troubleshooting Guide,* and the *Internetworking Design Guide.* For current Cisco Press titles and other information, go to Cisco Press online at this URL:

  http://www.ciscopress.com

- *Packet* magazine is the Cisco quarterly publication that provides the latest networking trends, technology breakthroughs, and Cisco products and solutions to help industry professionals get the most from their networking investment. Included are networking deployment and troubleshooting tips, configuration examples, customer case studies, tutorials and training, certification information, and links to numerous in-depth online resources. You can access *Packet* magazine at this URL:

  http://www.cisco.com/go/packet

- iQ Magazine is the Cisco bimonthly publication that delivers the latest information about Internet business strategies for executives. You can access iQ Magazine at this URL:

  http://www.cisco.com/go/iqmagazine

- Internet Protocol Journal is a quarterly journal published by Cisco Systems for engineering professionals involved in designing, developing, and operating public and private internets and intranets. You can access the Internet Protocol Journal at this URL:

  http://www.cisco.com/en/US/about/ac123/ac147/about_cisco_the_internet_protocol_journal.html

- Training—Cisco offers world-class networking training. Current offerings in network training are listed at this URL:

  http://www.cisco.com/en/US/learning/le31/learning_recommended_training_list.html

Introduction

The Cisco Element Management Framework CORBA[1] Gateway is part of an overall Integration Strategy that addresses a general requirement from the market for open, standards-based systems. Within this general requirement, specific requirements from the market for element management systems are for:

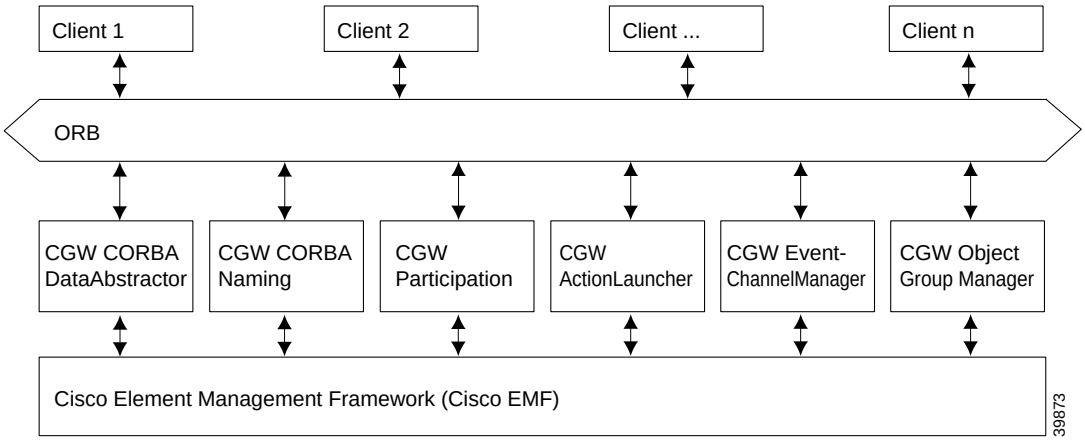
- Integration with higher level Network and Service Management Systems for the purpose of participating in an automated provisioning process.
- General purpose access to objects and methods which reside on Cisco EMF, for example initiate loop-back test for an xDSL line, or check status of a specific subscribers connection.
- Integration with Trouble Ticketing systems to enable faults in Cisco EMF to be taken in as part of a carrier's full operational processes.
- Integration of a 3rd party reporting tool with Cisco EMF to enable operational and management reports to be generated, for example Inventory Report.
- Access to Cisco EMF's database for ad-hoc access to data that is stored within Cisco EMF.

Cisco EMF CORBA Gateway Architecture

The Cisco EMF CORBA Gateway consists of a small number of CORBA servers, which are themselves clients of Cisco EMF. These provide a language- and platform- independent interpretation service for 3rd party developers wishing to write clients to access the powerful functionality of Cisco EMF. Access control to these servers is handled by a CorbaGatewayManager.

1. CORBA is a registered trademark of the OMG.

Figure 1-1 Cisco EMF CORBA Gateway



Basic Concepts

This section provides an overview of the concepts in Cisco EMF which should be understood prior to starting development of any Cisco EMF-based integration solution using the CORBA Gateway. These concepts result in benefits including:

- A future-proofed, flexible software architecture which can be extended to meet your application requirements
- A protocol independent object model which forms the foundations of Cisco EMF applications
- A mechanism for ensuring that all applications, no matter who develops them, inter-operate seamlessly with each other and the standard Cisco EMF applications. This is particularly suitable for the multi-vendor, multi- technology environment which exists in carriers today.

We assume that you are a software development professional with both object oriented software technology expertise and network management skills. In particular, we assume you are familiar with CORBA, and the basic concepts of network management systems integration.

Developing CORBA applications on Cisco EMF will therefore involve a number of techniques and concepts which you will be familiar with, and a number which are unique to Cisco EMF.

This chapter includes the following information:

CORBA

CORBA (Common Object Request Broker Architecture) is a distributed object architecture that allows objects to interoperate across networks regardless of the language in which they were written or the platform on which they are deployed. Thus CORBA allows developers to write applications that are more flexible and future-proof, to wrap legacy systems, and to code in the language they know best.

The Object Request Broker (ORB) is the middleware that handles the communication details between the objects. The CORBA 2.0 standard, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can communicate using a common protocol. CORBA is increasingly being used in the telecommunications field to provide connectivity between different applications and protocols.

Each CORBA object is basically broken into two pieces, a client-side proxy and a server-side implementation. The server side implementation contains the application logic. The client-side proxy is a proxy for the server-side object, and forwards method calls to the server-side implementation. On instantiation, the client-side proxy binds to the server-side implementation to tie together the two pieces. For further information on CORBA and development of CORBA applications, please refer to Other Resources, page A-1.

The ORB chosen for the development of the Cisco EMF CORBA servers is Orbix from Iona Technologies.

Orbix is CORBA 2.0 compliant, and so the CORBA Gateway is interoperable with CORBA 2.0 compliant ORBs from other vendors. Developers of client applications may choose to use Orbix, OrbixWeb or any other compliant ORB. The examples in this manual use Orbix as the client-side ORB, but instructions for connecting clients using a different ORB will be given in the Tutorial section.

Interface Definition Language

CORBA Interface Definition Language[1] (IDL) is an abstract programming language used to define the interfaces to the CORBA servers. IDL defines the functionality offered by the servers without specifying anything about the underlying implementation of those servers.

To develop a client application, the developer must take the IDL file provided with the server, and parse it using an appropriate IDL compiler. IDL compilers are readily available, both commercially and as freeware, to serve most languages and platforms. The generated stub code can then be incorporated into the client application.

The CORBA Naming Service

The CORBA Naming Service is one of the services defined by the OMG as part of the CORBA standard. Servers can register themselves with the Naming Service, and their object reference is mapped against a hierarchical naming structure. Clients can then easily locate the server by asking the Naming Service to lookup the appropriate name.

The IDL file defining the Naming Service is supplied by the OMG and is therefore standard across all ORBs, but there are various implementations provided by different ORB vendors. The IDL file should be equivalent, regardless of which ORB is being used. The Cisco EMF CORBA Gateway uses the Orbix Naming Service from Iona Technologies, and the IDL file (called NamingService.idl) supplied by Iona can be examined in the Developer Toolkit.

If using a different ORB for the client side, the NamingService.idl file should be compiled using the client-side IDL compiler to generate the NamingService.hh file for inclusion, and stubs for linking in with your client code.

1. CORBA Interface Definition Language is a registered trademark of the OMG.

Understand the Object Model of your Element Manager

Cisco EMF is based on a flexible dynamic object model which enables you to model a complex network in object oriented terms, with each network element modeled as a network object. This provides both a user and network centric view of the network topology and equipment. This is in contrast to the effective 'one dimensional' view often provided by SNMP MIBs. Such 'dynamic object modeling' typically involves a process of mapping attributes from the local Cisco EMF database and SNMP sources into Cisco EMF objects.

Object Identifiers

In Cisco EMF, it is not just managed network objects that are objects. Every attribute, object class, event, containment tree etc. is an object, identified by a unique, generic, and opaque Object Identifier, or ObjectID. The type of the object (class, attribute, etc.) is encoded within the ObjectID. The ObjectID is modeled in IDL by the ATL_OBJ::ObjectID structure.

The ObjectIDs for known named objects can be retrieved using the AtlNaming interface, see Chapter 6, "Naming and Identifying Cisco EMF Objects". Typically, for efficiency, a client application will retrieve the ObjectIDs for the containment trees, object classes and attribute names it requires at the start of the program and store them for later use.

Cisco EMF CORBA Gateway Design

A client of the Cisco EMF CORBA Gateway must first make contact with the CORBA GatewayManager. This server controls access to the other Cisco EMF servers by verifying the client's userID and password. Cisco EMF permits users to access only the services for which they have been authorized. Refer to the *Cisco EMF User Guide* for further details of Access Control. Identification of the user allows audit trail facilities to be provided.

The services which will probably be of most interest to external clients are:

- Participation—Allows external clients to initiate and participate in object lifecycle operations such as creation and deletion
- ActionLauncher—Enables the client to invoke actions on element managers
- Event Channel Manager—Allows clients to receive Cisco EMF events
- Object Groups—Enables clients to create, change and query the object groups that exist inside of Cisco EMF

They are supported by two utility services:

- AtlNaming—Used to translate between network object names and Cisco EMF ObjectIDs
- DataAbstractor—Allows Cisco EMF object attributes to be retrieved and set

Further information on the Cisco EMF servers can be found in the *Cisco EMF Developer Concepts Manual*. However, it must be borne in mind that not all concepts described may be applicable to the CORBA developer.

Cisco EMF CORBA Server Design

The Cisco EMF CORBA Gateway design adopts a ‘service-oriented’ approach, that is to say, there are a small number of CORBA server objects which provide the functionality to manipulate the large number of network managed objects. Managed objects are identified and manipulated by opaque identifiers, and are not exposed as CORBA objects in their own right. This approach was taken for reasons of scalability and speed, since instantiating every managed object as a CORBA object would impose a heavy overhead thus limiting the system’s scalability without adding functionality.

Asynchronous Calls

Cisco EMF uses asynchronous communications almost exclusively because synchronous calls which initiate time-consuming processing on the servers can potentially block for long periods of time. By avoiding these processing bottlenecks, this asynchronous design has proved quicker and more robust than synchronous versions. The CORBA Gateway API also heavily relies on asynchronous calls to reflect this underlying design philosophy.

For each interface which uses asynchronous calls, there is a corresponding callback interface, and for each asynchronous request call, there is a callback method in the callback interface. This callback method is used to reply to the user, returning the result of the request. The object reference for the object which instantiates the callback interface is passed as a parameter to the request call, along with a userdata parameter, which is returned to the user with the reply callback, and can be used to correlate the requests with their replies.

The Participation Framework

The Cisco EMF Participation Framework has been developed to allow arbitrary clients to initiate and participate in arbitrary object management operations within Cisco EMF. The Framework is completely generic and data-driven, so although its main use at this time is for object deployment, it could be used for other types of object lifecycle operation in the future.

There are three types of actors involved in a Participation scenario:

- ParticipationCoordinator
- Participant(s)
- Initiator

The Participation Coordinator is an Cisco EMF server which oversees the Participation operations.

The phases of the operation cycle are represented by a number of participation levels. The coordinator ensures that all the levels have been processed in the correct order for successful completion of the operation.

The data which defines the operation to be carried out is encapsulated in a participation context. The initiator creates the context and fills it with the appropriate data, before submitting it to the coordinator to initiate the operation. The initiator also specifies the start and end levels of the operation. An application which creates new managed objects in Cisco EMF is an example of an initiator.

Each participant registers with the coordinator to indicate its intention to participate in operations at a particular level. An inventory reporting application which needs to monitor managed object lifecycle events such as object creation and deletion would be an example of a participant. As the operation proceeds and reaches a level at which there is a registered participant, the coordinator will pass the

context to that participant, which can then carry out its processing or monitoring (including modifying the contents of the context). When ready, it passes the context back to the coordinator. The coordinator then passes the modified context on to the next level and registered participants in the cycle.

If the operation encounters an error condition and fails, it will start to revisit the earlier levels in reverse order, giving the participants the opportunity to undo their processing.

Upon reaching the final level, the initiator is informed of the successful completion of the operation, and is passed the final contents of the context.

DataAbstractor

The data abstractor is used whenever you need to traverse a containment view. It is also used for getting and setting attributes. Hence the primary role of the data abstractor is for provisioning and inventory management. In the future the DataAbstractor is also going to be used for fault management - raising, acknowledging and clearing faults, and performance management - to get historical performance data.

Object Group Server

The CORBA Object Groups API allows clients to manipulate Cisco EMF Object Groups.

An object group is simply a collection of objects which are related in some way. They may all be the same type of equipment or all belong to the same customer.

Object groups can be built either manually or by building a query. Some Cisco EMF subsystems may also build object groups which may be visible and usable by the Cisco EMF user.

The CORBA Object Group server provides CORBA clients with the ability to manipulate the Object Groups stored in Cisco EMF. CORBA clients can create and delete object groups using the CORBA Deployment interface. Also, clients can add or remove objects from groups and iterate over the contents of object groups.

A good example of the use of the CORBA Object Group server is obtaining all the CISCO 6260 devices being managed and then using the CORBA ActionLauncher to perform an action on each of them.

Further information on Object Groups may be found in the *Cisco EMF Developer Concepts Manual*.

Event Channels

The Event Channel Service can be used for fault management to notify an EMS of a new alarm, or the change of status of an existing alarm. It can also be used in provisioning and inventory applications to notify the higher level system of new network elements or insertion or removal of modules in a NE.

The Cisco EMF CORBA Gateway uses the standard OMG Notification Service to export internal events to interested CORBA-based clients. The clients register with notification channels, expressing their interest in particular event types available on the channels and optionally specifying a filter to be used. The events exported allow clients to keep track of changes happening within Cisco EMF, for example:

- Object Creation /Deletion
- Metadata changes
- Alarms raise/cleared
- Attribute changes

- SNMP Traps received / mapped
- etc

The Events are sent via the CORBA standard's Notification Service, this service has full filtering capabilities and implements various optimizations

ActionLauncher

In some Element Managers provisioning of services to subscribers is done using user actions. The ActionLauncher allows CORBA clients to invoke actions in Cisco EMF. Actions are high-level commands which perform complex tasks on an element manager. These are the tasks which are typically launched by the end user clicking a button on the EMS GUI. For example -

- Connecting a new subscriber to a service
- Initiate a software download
- Initiate a sub-rack discovery
- Call APIs on an EMS
- Set/Move the state of object in an EMS

Tutorials: Getting Started

This section takes the reader through the steps required to build a simple CORBA client to access a CORBA Gateway server. It also includes hints on how to achieve some common tasks using the Cisco EMF CORBA interfaces.

Tutorial: a Simple Cisco EMF CORBA Client

This section shows the key components of a very simple Cisco EMF client developed in C++ using Iona Technologies' Orbix or Java using Iona Technologies' OrbixWeb. The DataAbstractor is used as the example server since it has a very simple and typical interface.

Note Also, most of the error checking and exception handling has been omitted for clarity. Please refer to your Orb vendor's literature for full details of appropriate implementation in your environment.

This section includes the following information:

Generate Stubs and Skeletons from the IDL using C++

IDL files define the interfaces to the Cisco EMF CORBA Gateway servers. These files can be found in the Cisco EMF CORBA Gateway installation area (see the *Cisco EMF CORBA Gateway Installation Instructions* for details). If the client is to be developed on a platform other than that of the servers, then the IDL files need to be copied to the client platform.

For your client application to use the Cisco EMF CORBA Gateway, it must be able to access the functionality of the interfaces defined in the IDL files. IDL compilers are provided by ORB vendors to translate the IDL into stubs and skeletons in the required language which are then used in the development of distributed applications. For example, using the Orbix IDL compiler, DataAbstractor.idl can be compiled by the following command:

idl -B -A -Iidldir DataAbstractor.idl

This will produce three files:

- DataAbstractor.hh a header file which should be included in the client program
- DataAbstractorC.cc a C++ source file to be compiled and linked with clients of the servers defined in DataAbstractor.idl (also known as the stub file).
- DataAbstractorS.cc a C++ source file to be compiled and linked with implementations of the servers defined in DataAbstractor.idl (the skeleton file).

Optionally, the -S flag can also be used, in which case two extra files are generated:

- DataAbstractor.ih contains templates to aid development of the header files for the implementation classes
- DataAbstractor.ic contains templates to aid development of the code for the implementation classes

Where idldir is the directory where the Cisco EMF IDL files have been installed on your system. This is required since the Participation.idl file includes other IDL files for basic data type definition.

Although we are developing a client application, it is the skeleton file which must be linked in, since the client must also act as a server in order to handle callbacks. This will be discussed in more detail in The Client is also a Server, page 3-7.

The client code must include the header files generated by the IDL compiler, these will typically include files corresponding to the CorbaGatewayManager, the CORBA Naming Service and the Cisco EMF server(s) to be used. Thus, in this example, the files to be included are: CorbaGatewayManager.hh and Naming.hh and DataAbstractor.hh.

Generate Stubs and Skeletons from the IDL using Java

Just as in the previous example, the IDL files describing the Cisco EMF CORBA Gateway and the DataAbstractor server must be compiled using an appropriate IDL compiler to produce the Java classes which will be used in the development of the client. The complete OMG IDL to Java mapping specification can be found on the OMG website, in your ORB vendor's documentation and elsewhere.

Using the OrbixWeb IDL compiler, DataAbstractor.idl can be compiled by the following command, enter:

idl -N -jO. -I*idldir* DataAbstractor.idl

This will map the IDL definitions to Java classes and packages, in a directory structure corresponding to the IDL modules. An IDL interface called X will map to (among others) a Java interface also called X used by the client, and an abstract class called _XImplBase, which the developer of the server will use to implement the X functionality.

The client application must import the classes or packages corresponding to the CorbaGatewayManager, the CORBA Naming Service and the Cisco EMF server(s) to be used. Thus, in this example, the import statements at the head of the client application will include the following:

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.* ;
import ATL_OBJ.*;
import ATL_DABS.*;
import ATL_GATE.*;
```

CORBA Initialization

We can now start to write the client application. Before any CORBA calls can be made, we must initialize the ORB:

C++

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "Orbix”);
```

Java

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

Using the Orbix Naming Service to Locate the Cisco EMF CORBAGatewayManager

Resolve the Orbix Naming Service root context:

C++

```
CORBA::Object_var ns =
    orb->resolve_initial_references("NameService”);
CosNaming::NamingContext_var nsRootContext =
    CosNaming::NamingContext::_narrow(ns);
```

Java

```
org.omg.CORBA.Object ns =
    orb.resolve_initial_references("NameService");
NamingContext ncontext = NamingContextHelper.narrow(ns);
```

then obtain a reference to the Cisco EMF CorbaGatewayManager object by interrogating the CORBA Naming Service:

C++

```
ATL_GATE::CorbaGatewayManager_var corbaGatewayManager_var;

CosNaming::Name_var name = new CosNaming::Name(2);
name->length(2);
name[0].id = CORBA::string_dup("atl");
name[0].kind = CORBA::string_dup("");
name[1].id = CORBA::string_dup("CorbaGatewayManager");
name[1].kind = CORBA::string_dup("ServerObject");
```

```
CORBA::Object_var tempObj = nsRootContext->resolve(name);
corbaGatewayManager_var gateway_var =
   ATL_GATE::CorbaGatewayManager::_narrow(tempObj);
```

Java

```
NameComponent nc = new NameComponent(
        "CorbaGatewayManager",
        "ServerObject");
NameComponent name[] = {nc};
org.omg.CORBA.Object tempObj = ncontext.resolve(name);
CorbaGatewayManager gateway =
        CorbaGatewayManagerHelper.narrow(tempObj);
```

Note The `resolve_initial_references` call will only work for objects implemented in the same ORB environment, in this case, Orbix. For clients written using other ORBs, the Orbix Naming Service root context has to be resolved by use of stringified IORs. Refer to Stringified Interoperable Object References, page 3-9.

Connecting to an Cisco EMF CORBA Service

The CorbaGatewayManager gives access to the other Cisco EMF servers after authentication of a supplied userID and password. For example, here is the IDL fragment showing the call to access the servers:

```
Object get(
        in string serverName,
        in string userID,
        in string password);
```

So we can now obtain a reference to the DataAbstractor server object by making the following call :

C++

```
CORBA::Object_var tempServerObj = gateway_var->get(
        “DataAbstractor”,
        “admin”, “admin”);
```

Java

```
org.omg.CORBA.Object dabsTempObj = gateway.get(
        "DataAbstractor",
        "admin","admin");
```

This call will throw an exception if an invalid server name is given, or if the user and password are not authorized.

Note Refer to the *Cisco EMF User Guide* for details of authentication of userIDs and passwords for access to services.

By analogy with the Naming Service, the returned object reference must be narrowed to the correct type, in this case DataAbstractor:

```
ATL_DABS::DataAbstractor_var dataAbstractor_var =
    ATL_DABS::DataAbstractor::_narrow(tempServerObj);
```

Invoking Methods and Receiving Replies through Asynchronous Callbacks

The client can now use this DataAbstractor object to make requests. However, since all the calls on this server are asynchronous, the client needs to provide an implementation of the DataAbstractorCallback interface in order to be able to receive the replies. An instance of this implementation class is passed as a parameter in every request. For example, here is the IDL fragment showing the DataAbstractor method which gets the values of attributes:

```
interface DataAbstractor
{
    short getAsync(
            in ATL_OBJ::ObjectAttributeList requests,
        in DataAbstractorCallback daCallback,
        in unsigned long userData);
};
```

where

- `requests` is the list of attributes to be retrieved, encapsulated in an `ATL_OBJ::ObjectAttributeList` data structure
- `daCallback` is the instance of the DataAbstractorCallback implementation object
- `userData` is an arbitrary integer, chosen by the user, which will be returned with the reply. The user may wish to utilize this to correlate requests and replies.

Developing and Instantiating a Callback Object

The DataAbstractorCallback interface contains the definition of the corresponding reply method:

```
interface DataAbstractorCallback
{
            oneway void getCB(
                in ATL_OBJ::ObjectAttributeList results,
                    in unsigned long userData);
};
```

where

- `results` is the returned list of attributes, containing both successes and failures
- `userData` is the returned user data

The client application developer must write an implementation of the DataAbstractorCallback interface, including this method:

C++

```
void DataAbstractorCallback_i::getCB (
            const ATL_OBJ::ObjectAttributeList& results,
            CORBA::ULong userData,
            CORBA::Environment &IT_env)
{
            cout<<"[DataAbstractorCallback_i::getCB]"<<endl;
            cout<<" results have "<<results.length()<<"
            elements"<<endl;

            ....... etc. .....
}
```

Java

```
import ATL_OBJ.*;
import ATL_DABS.*;

class DataAbstractorCallbackImpl extends _DataAbstractorCallbackImplBase
{
    public void getCB(
        ATL_OBJ.ObjectAttributeListEntry[] results,
        int  userData)
    {
        ....... etc. .....
    }
        ....... etc. .....
}
```

Having written an implementation of this callback interface, an object of this type must be instantiated in the client application. For example:

C++

```
//        for the asynchronous calls, we also need to
//        create a callback object:

dataAbstractorCallbackObject = new
        DataAbstractorCallback_i();
```

Java

```
//        for the asynchronous calls, we need to create a
//        callback object and connect it to the ORB:
DataAbstractorCallbackImpl dataAbstractorCallbackImpl =
          new DataAbstractorCallbackImpl();
orb.connect(dataAbstractorCallbackImpl);
```

Making an Asynchronous Call

The client application now has all that it needs to make calls on the DataAbstractor server. For example:

C++

```
// Create an outbound ObjectAttributeList
ATL_OBJ::ObjectAttributeList_var  objAttrList =
    new ATL_OBJ::ObjectAttributeList;
objAttrList->length(1);
            ....... fill list with data .....

dataAbstractor_var->getAsync(
        *objAttrList,
        dataAbstractorCallbackObject,
        0);
```

Java

```
// Create an ObjectAttributeList with 1 entry
ATL_OBJ.ObjectAttributeListEntry[]  objAttrList =
    new ATL_OBJ.ObjectAttributeListEntry[1];

// fill this objAttrList with data...
        ..... etc....

dataAbstractor.getAsync(
        objAttrList,
        dataAbstractorCallbackImpl,
        0);
```

Note The try... catch exception handling has been omitted here for clarity, however it is an essential part of a robust distributed application.

The Client is also a Server

Many of the Cisco EMF CORBA Gateway interfaces are asynchronous. An asynchronous interface means that to send the results of invocations, the server needs to be able to call back to the clients. This is achieved by callback interfaces being implemented by the client. These callback interfaces are defined in the IDL for each asynchronous interface.

A client application which implements an interface is acting as a server and must be linked with the appropriate server code generated from the IDL. Also, to process callbacks it must also be able to receive incoming CORBA events. In an Orbix client, this can be achieved by calling CORBA::Orbix.impl_is_ready(), or more explicitly asking Orbix to process incoming events with a call such as CORBA::Orbix.processEvents(). If the application needs to be able to respond to other input (keyboard, GUI, events from other processes) as well as the incoming CORBA messages, then some careful coding is required to share processing time between the CORBA event loop and the other input event loops.

Exception Handling

Remote method calls can throw exceptions. It is good programming practice to catch these exceptions, for example, to enclose the calls in a try .... catch clause, as appropriate in the programming language used. CORBA calls can throw both system exceptions (for example, when there is communication failure between client and server), and user exceptions which are defined by the server developers and relate to specific problems encountered during the method invocation. The client developer can check the user exceptions which could be thrown by a particular method by consulting the IDL. Catching all possible exceptions and handling them appropriately is essential for a robust client application.

Running a C++ Client

After compiling and linking the client code it should be ready to run. The Cisco EMF core servers and CORBA servers should already be installed and running, and the Orbix daemon running before attempting to run the client.

For help with possible configuration and communication problems, see Chapter 12, “Troubleshooting and Helpful Hints”.

Running a Java Client

After compiling the Java client with an appropriate Java compiler compatible with the Java ORB, it should be ready to run. The Cisco EMF core servers and CORBA servers should already be installed and running, and the Orbix daemon running, before attempting to run the client. The CLASSPATH environment variable should be checked to make sure that all the appropriate Java system classes and the ORB classes are accessible.

Tutorial: How to Access the Orbix Naming Service from a Client using another ORB

Under the current version of the CORBA standard, the CORBA Naming Service is standardized and interoperable across all ORBs. However, the client has to obtain an initial reference to the Naming Service object. This is easy for clients developed under the same ORB, but requires a little more effort for other clients. These clients must use a stringified IOR to bootstrap the connection to the Naming Service, as described below. In the future, this workaround should not be necessary as ORB vendors adopt the recently (March 1999) defined Interoperable Naming Service bootstrap protocol.

This section includes the following information:

Stringified Interoperable Object References

Stringified Interoperable Object References (IORs) are used to transmit object references between different ORBs. A stringified IOR is non-human readable ASCII string, typically a few hundred bytes long, which starts with the characters “IOR:”. Here is an example:

```
IOR:000000000000002049444c3a436f734e616d696e672f4e616d696e67436f6e746578743a312e3000000000
0100000000000000040000100000000000066a6f6e6573000639000000 2c3a5c6a6f6e65733a4e533a3a3a49523a
436f734e616d696e675f4e616d696e67436f6e7465787400000000
```

There are two types of IOR:

- A transient IOR is valid only as long as the server process that contains the object stays active.
- A persistent IOR always identifies a particular remote object, irrespective of how many times a server is shut down or times out. Essentially, a persistent IOR points to a remote CORBA object that conceptually always exists.

Whether a server has a transient or persistent IOR depends on its startup configuration, that is whether it starts on a well known port, or uses the Orbix daemon as a locator. Orbix server configuration is beyond the scope of this manual, refer to the Orbix documentation for further details.

Note The IOR for the Gateway and the CORBA Notification Service are published in the directory *<CEMF_ROOT>***/config/ior/**

Destringify the IOR in the Client

There are two basic ways of obtaining the IOR on the client side:

- Read in the IOR string directly from the IOR text file every time
- The IOR could be hard-coded into the text (not for transient IORs!)

Having obtained the IOR string, the development of the client follows the same pattern as that of the simple client described in the tutorial above, with only one small change as follows. Remove the line of code which obtains the reference to Naming Service by calling resolve_initial_references(), and replace it with a call to string_to_object(), which destringifies the IOR to obtain an object reference. For example:

C++

```
char* ior_string = "IOR:00000000000000002049444...etc...";
CORBA::Object_var ior_object =
    orb->string_to_object(ior_string);
```

Java

```
String ior_string = "IOR:00000000000000002049444...etc..";
org.omg.CORBA.Object ior_object =
    orb.string_to_object(ior_string);
```

This object can then be narrowed to resolve the root context, and the rest of the processing is as described in the basic tutorial.

CHAPTER 4

Accessing the Cisco EMF Servers

CorbaGatewayManager API

Note See Chapter 2, "Basic Concepts" for an overview of the concepts in Cisco EMF which should be understood prior to starting development of any Cisco EMF-based integration solution using the CORBA Gateway.

The CorbaGatewayManager controls the access of external clients to the CORBA Gateway servers after authentication of a userID and password. The CorbaGatewayManager interface is shown below:

```
interface CorbaGatewayManager
{
    Object get (in string serverName,
        in string user,
        in string password)
    raises (ATL_EXCEPT::invalidServerName,
        ATL_EXCEPT::authorisationFailure);
};
```

where the exceptions are defined as:

```
module ATL_EXCEPT
{
    exception invalidServerName
    {
        string reason;
        string serverName;
    };
    exception authorisationFailure
    {
        string reason;
        string user;
        string password;
    };
};
```

This get function is used to access all the CORBA servers, which are identified by a known serverName. If an invalid or unknown serverName is given, then the invalidServerName exception is thrown. The user and password parameters are used to identify the CORBA client as an authorized Cisco EMF user. If the user and password given are not authorized to use the service requested, the authorisationFailure exception is thrown. Refer to the *Cisco EMF User Guide* for more details of Access Control.

After successful authentication of the client's credentials, each method returns an object reference to the appropriate CORBA Gateway server, which after narrowing to the appropriate object type, can then be used as described in the following sections.

Creating Cisco EMF Objects

The basic concepts of the Cisco EMF Participation Framework have already been introduced in The Participation Framework, page 2-4. The Participation Framework and Deployment Model will be discussed in greater detail in this section.

The Participation Framework provides a closed-loop control mechanism for sending data round an arbitrary number of interested processes. It provides a flexible mechanism for extending Cisco EMF during tasks such as deployment. Participants can add to the deployment data, or merely be informed when it is at a particular stage.

There are three types of actors involved in a Participation operation:

- The ParticipationCoordinator (which is provided as a Cisco EMF CORBA Gateway Server)
- The Participant(s) (which can be implemented as CORBA clients)
- The Initiator (which can be implemented as a CORBA client)

The Initiator creates and submits a ParticipationContext, containing all the data required to drive the operation, to the ParticipationCoordinator.

Participants register their interest in the operation at one or more participation levels. The context is passed to the Participant registered at each level in turn for processing. When the context has successfully visited all levels for this operation, the context is complete.

The ParticipationCoordinator manages and controls all the participation interactions.

The sequence of events is illustrated by a UML sequence diagram in Figure 5-1 and Figure 5-2.

Figure 5-1 Initiator Sequence

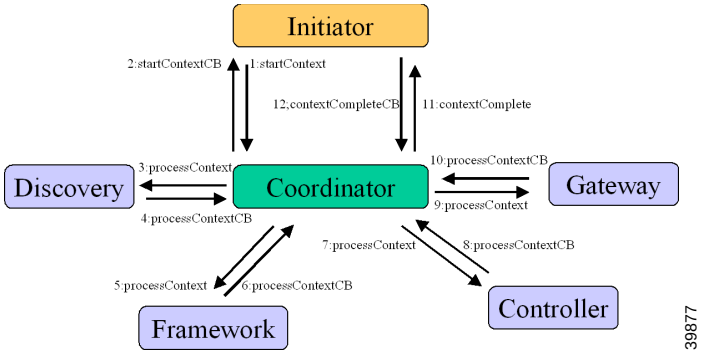
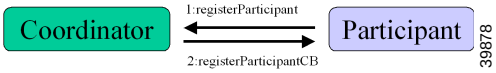


Figure 5-2 Participant Registration Sequence



A CORBA client can act as a Participant, an Initiator, or both, as explained in the following example. A CORBA client application wishing to both initiate and participate in a Participation cycle would perform the following tasks:

Step 1 Obtain a reference to the ParticipationCoordinator object.

Step 2 Create implementation objects for the Participant and Initiator interfaces.

Step 3 Register the Participant object at one or more participation levels.

Step 4 Create a context (for example, a DeploymentContext), and insert data into it as appropriate.

Step 5 Start the operation by submitting the context to the ParticipationCoordinator, specifying appropriate start and stop levels for the operation, and supplying the Initiator object for callbacks.

Step 6 Process incoming CORBA messages, and expect to receive:

(Participant) registerParticipantCB confirmation of participant registration at each level

(Initiator) startContextCB confirmation of context successful start

(Participant) processContext allows participation at each level reached. Remember to call processContextCB!

(Initiator) contextComplete successful completion of the context when all levels have been processed. Remember to call contextCompleteCB!

Step 7 Finally, deregister the Participant from all the registered levels, and process the incoming deregisterParticipantCB calls.

The Deployment Model

The most important use of the Participation Framework at this time is for the creation and deletion of Cisco EMF objects - a process known as deployment. The data and interface definitions to support deployment are contained within the ATL_Deploy module.

A CORBA client may want to take part in deployment for various reasons. For example, a client may need to keep the object models held in Cisco EMF and an external system consistent. In this case, it would need to:

- Create or delete objects in Cisco EMF, to keep in step with changes to objects in the external system. It would do this by acting as an Initiator of a DeploymentContext when changes occurred in the other system.
- Create or delete objects in the external system, to keep in step with changes to objects in Cisco EMF. It would do this by acting as a Participant registered to monitor the Cisco EMF object creation and deletion participation levels.
- Create or delete object groups
- Create or delete containment views

Participation Levels Used in Deployment

Deployment operations proceed in a series of phases. Each phase is further split up into separate participation levels, each for a different type of Cisco EMF object. This is to ensure that Cisco EMF objects get created in the correct order, according to their type. For example, containment tree objects should be created before managed objects. Likewise, the different types of Cisco EMF object must be deleted in the correct order too.

Deployment creation has four main phases:

1. Definition—Participants get the chance to add more ObjectDefinitions to the context or expand existing ObjectDefinitions with new attributes or containments.

   The Definition phase is further subdivided as follows:

   – Addition—Participants are allowed to add new managed object definitions. For example an element manager controller might add several modeling objects below an agent.

   – Expansion—Participants are allowed to add information to managed object definitions. For example, the auto discovery server might ensure that every IP addressable object has a valid network containment path, to keep its IP model intact.

2. Creation—The objects defined in the context are created in Cisco EMF. Exactly one participant for each object type is allowed to register in this phase - the behavior is undefined when more than one participant per object type is registered.
3. Action—Participants get the chance to initialize the functionality of the newly created objects. For example, an element manager controller might start a state machine for a managed object, interrogate the device and then create virtual objects in a recursive provisioning cycle.
4. Event—Deployment creation events are raised.

Each phase of deployment is carried out in strict order of the category of objects being created. Trees get created first, followed by groups, managed objects and finally maps, each level being denoted by a unique numeric value, which is defined in the reference section at the end of this section.

The phases of deployment deletion are similarly defined, except in this case, action is before deletion:

- Definition—Participants get the chance to add more ObjectDefinitions to the context or expand existing ObjectDefinitions with new attributes or containments.
- Action—Participants get the chance to terminate functionality of objects before they are destroyed. For example, an element manager controller might stop a state machine for a managed object and remove some configuration from a device it manages.
- Deletion—The objects defined in the context are deleted. Exactly one participant for each object type is allowed to register in this phase.
- Event—Deployment deletion events are raised.

As with creation, the deletion phases are further subdivided by category of object.

Note The Participation Framework does not enforce the behavior described above but you must follow these guidelines to get well-defined behavior.

Error Handling

The Cisco EMF Participation Framework will implement complete success or complete failure semantics. If the Framework Participant responsible for creating an object fails to create it, then it will undo all that it had successfully completed up to that point. This behavior is supported by the Participation Framework which provides rewind functionality. Rewind means that the context will go backwards through all of the levels visited so far with an undo flag set. Each participant can indicate if it is interested in processing a rewinding context when it registers at a given level. For example, all of the Framework Participants will register to be called during rewind so that managed objects can be deleted if a subsequent abstraction creation fails. The EMS Controller is also interested in rewind, in this case so that it can stop state machines.

Deployment Data

Deployment data is represented by a DeploymentContext. The DeploymentContext has a layered structure, each layer is used to store ObjectDefinitions for a single category of object. For example, ObjectDefinitions for managed objects are in one layer while ObjectDefinitions for containment trees are in another.

An ObjectDefinition has the following associated data items:

- type—The category of object to be created
- class—The ObjectID of the class of this object, for example, the ObjectID for site or bay class.
- containments—A vector of containment relationships for this object
- attributes—Attributes that should be set on the object during creation.
- failures—A list of attributes which failed at creation. This list will remain empty unless attribute failures cause creation to fail.

- participant attributes—A list reserved for participants to add attributes which are private to the participant.
- status—The status of this ObjectDefinition. The status can have the values detailed below.
- ObjectDefinition ID—A unique ID for this ObjectDefinition within the scope of the context. This allows an ObjectDefinition to refer to other ObjectDefinition in the Context before any of the objects have been created. This is necessary for building containment relationships, etc.

As well as the ObjectID which uniquely identifies it within the DeploymentContext, each ObjectDefinition has two extra ObjectIDs associated with it, which are used during the deployment process:

- allocatedID—This ObjectID is invalid prior to object creation and is assigned when the object is successfully created, uniquely identifying the object.
- ObjectIDforObjectDefinition—This can be used in place of the real objectID within any attribute values and containment paths of an ObjectDefinition. This allows cross-referencing of ObjectDefinitions before real objectIDs are available. The creation platform substitutes these ObjectIDs within attribute values for the real ObjectIDs immediately before calling create APIs.

Each ObjectDefinition in the context also has a status value, which can take the following values:

- unprocessed—This ObjectDefinition has not been processed.
- okay—This ObjectDefinition has been processed successfully. The object will exist unless rewind has taken place.
- error—This ObjectDefinition could not be created. Containment failures will have their ContainmentSpec status values changed to indicate failure. Attribute failures will be put on the failure list.

Each ObjectDefinition has a list of Containment Specs and Attributes. The ContainmentSpec list specifies the containment relationships which the object is to be given. Each ContainmentSpec includes a textual name for the object in that relationship. The Attributes specify the Cisco EMF attributes which will be set upon creation. Only the attributes which need to be set upon creation should be added to the Attributes list.

If any object defined in the DeploymentContext fails to be created for any reason, the whole deployment scenario will be rewound so that none of the objects in the scenario will exist at the end of the process. This means that either all of the objects get created successfully or none of them get created. The ObjectDefinitions in the DeploymentContext will contain information about the failure.

Participation and Deployment APIs

The Particiation IDL file defines the ATL_PART module, which defines data structures, interfaces and methods used in Participation Framework. The Deployment IDL file defines the ATL_Deploy module, which contains specific interfaces and data definitions required for deployment.

The Participant and Initiator Implementation Classes

All the interfaces defined in the ATL_PART module are implemented on the server except the Participant and the Initiator, which must be implemented by the client-side developer. This is due to the ‘dialogue’ which takes place between Participation clients and the ParticipationCoordinator. Every invocation to one is reciprocated by an asynchronous reply or acknowledgment. For example, a client acting as an Initiator calls startContext on the ParticipationCoordinator, which processes this request and responds

by calling startContextCB on the client to signal successful completion. The Participant and the Initiator client-side implementation classes are therefore necessary to allow the ParticipationCoordinator to call these acknowledgment and interaction methods.

Some of the methods to be implemented on these objects are callbacks confirming the successful completion of an action requested by the client (for example startContextCB). The client does not need to do anything special in these methods, they are for the client's information. Other methods are actions initiated by the ParticipationCoordinator (for example processContext). In these cases, after finishing its processing, the client must reciprocate by calling the appropriate callback (for example processContextCB), since the ParticipationCoordinator is waiting for this confirmation to proceed.

The Participant interface to be implemented is defined as follows:

```
interface Participant
{
    oneway void registerParticipantCB(
        in ParticipationStatus status,
        in unsigned long userData),

    oneway void deregisterParticipantCB(
        in ParticipationStatus status,
        in unsigned long userData);

    oneway void processContext(
        in long level,
        in ParticipationDirection dir,
        in CorbaParticipationContext cpc,
        in ParticipantCallback pCallback,
        in unsigned long userData);
};
```

The Initiator interface to be implemented is defined as follows:

```
interface Initiator
{
    oneway void startContextCB(
        in long contextID,
        in ParticipationStatus status,
        in ParticipationInfo pInfo,
        in unsigned long userData);

    oneway void stopContextCB(
        in ParticipationStatus status,
        in unsigned long userData);

    oneway void contextComplete(
        in long contextID,
        in CorbaParticipationContext cpc,
        in ParticipationInfo pInfo,
        in InitiatorCallback iCallback,
        in unsigned long userData);

    oneway void participationBroken(
        in long contextID,
        in CorbaParticipationContext cpc,
        in ParticipationInfo pInfo,
        in InitiatorCallback iCallback,
        in unsigned long userData);
```

```
    oneway void progressUpdate(
        in long contextID,
        in CorbaParticipationContext cpc,
        in ParticipationInfo pInfo,
        in InitiatorCallback iCallback,
        in unsigned long userData);
};
```

The client-side developer must write an implementation of one or both of these interfaces, as appropriate.

Note The first two methods in this interface are asynchronous callbacks, and so have 'CB' in their names. They are invoked on the Initiator by the ParticipationCoordinator to signal completion of the appropriate earlier call. For example, after the Initiator has invoked startContext on the ParticipationCoordinator, the incoming startContextCB response from the ParticipationCoordinator indicates completion of that task.

In contrast, contextComplete is also invoked on the Initiator by the ParticipationCoordinator, but does not have 'CB' in its name, since it is not a direct callback, but is called to inform the Initiator of an event in which it is interested, the completion of a participation context's processing by the registered Participants. Since the Initiator's acknowledgment of the receipt of this information is the final act in the context lifecycle, as shown in figure 1, the Initiator must invoke the corresponding callback on the ParticipationCoordinator interface: contextCompleteCB.

Similarly, the informational calls participationBroken and progressUpdate also expect the Initiator to respond by calling the corresponding callback on the ParticipationCoordinator interface.

Details of how to implement a server using the skeleton code generated from the IDL will be given in the user documentation for your chosen ORB. As an example, the C++ header file for an Orbix implementation class for this interface, using the BOAImpl approach, might look like this:

```
// Need to include the definition of the C++ class generated
// from the Initiator interface by the IDL compiler.

#include "Participation.hh"

// Initiator_i uses the BOAImpl approach to implement
// interface Initiator. That is, it inherits from the
// InitiatorBOAImpl class generated by the IDL compiler.

class Initiator_i: public ATL_PART::InitiatorBOAImpl
{
       public:
//  IDL Methods

    void startContextCB(
        CORBA::Long contextID,
        ATL_PART::ParticipationStatus status,
        const ATL_PART::ParticipationInfo& pInfo,
        CORBA::ULong userData,
        CORBA::Environment
        &IT_env=CORBA::default_environment)
        throw (CORBA::SystemException);

    void stopContextCB(
        ATL_PART::ParticipationStatus status,
        CORBA::ULong userData,
        CORBA::Environment
        &IT_env=CORBA::default_environment)
        throw (CORBA::SystemException);
```

```
    void contextComplete(
        CORBA::Long contextID,
        ATL_PART::CorbaParticipationContext_ptr cpc,
        const ATL_PART::ParticipationInfo&,
        ATL_PART::InitiatorCallback_ptr iniCallback,
        CORBA::ULong userData,
        CORBA::Environment
        &IT_env=CORBA::default_environment)
        throw (CORBA::SystemException);

    void participationBroken(
        CORBA::Long contextID,
        ATL_PART::CorbaParticipationContext_ptr cpc,
        const ATL_PART::ParticipationInfo& pInfo,
        ATL_PART::InitiatorCallback_ptr iniCallback,
        CORBA::ULong userData,
        CORBA::Environment
        &IT_env=CORBA::default_environment)
        throw (CORBA::SystemException);

    void progressUpdate(
        CORBA::Long contextID,
        ATL_PART::CorbaParticipationContext_ptr cpc,
        const ATL_PART::ParticipationInfo& pInfo,
        ATL_PART::InitiatorCallback_ptr iniCallback,
        CORBA::ULong userData,
        CORBA::Environment
        &IT_env=CORBA::default_environment)
        throw (CORBA::SystemException);
};
```

The CorbaParticipationContext

Since the Participation Framework is purely data-driven, everything required to drive a deployment operation is expressed by the data in the participation context. Contexts are defined in IDL as interfaces, derived from a base interface CorbaParticipationContext, which is defined in IDL as follows:

```
interface CorbaParticipationContext
{
        readonly attribute long contextID;
        attribute ParticipationStatus status;
        attribute ParticipationDirection direction;
        attribute boolean wantProgress;
        };
```

Where:

- contextID—Can be used by the participant to distinguish between different active contexts
- status—The current status of the process, which can be read or updated by the participant
- direction—Indicates whether the context is progressing forwards as normal, or is going in reverse, which indicates a roll-back after an error has occurred.

Specialized contexts to perform specific tasks, such as the deployment context, inherit from this base context.

Note Once a reference to a CorbaParticipationContext object has been passed in the call the **startContext** of a ParticipationCoordinator it cannot be used again by the client.

The DeploymentContext

DeploymentContext is a specialized context inherited from CorbaParticipationContext. Clients can obtain a DeploymentContext via the CorbaGatewayManager in the usual manner:

```
CORBA::Object_var tempObj;
try
{
    tempObj = corbaGatewayManager_var->get(
        "DeploymentContext",
        user,
        password);
}
catch(ATL_EXCEPT::invalidServerName &invalid)
{
        // do appropriate action for exception
}
catch(ATL_EXCEPT::authorisationFailure &author)
{
        // do appropriate action for exception
}

ATL_Deploy::DeploymentContext_var deploymentContext_var =
    ATL_Deploy::DeploymentContext::_narrow(tempObj);

if (CORBA::is_nil(deploymentContext_var))
{
        // deal with the error
}
```

This empty DeploymentContext is now filled with data before submission. The addObjectDefinition() method, defined below, allows the client to define the objects to be created in the Cisco EMF system during Deployment. A similar method exists for object deletion. The DeploymentContext interface also contains other data definitions and methods, omitted here for clarity.

```
interface DeploymentContext : ATL_PART::CorbaParticipationContext
{
    ATL_OBJ::ObjectID addObjectDefinition (
        in octet objectType,
        in ATL_OBJ::ObjectID classID,
        in ContainmentSpecVector containments,
        in ATL_OBJ::AttributeList attributes,
        in ATL_OBJ::AttributeList
        participantAttributes,
        in string name);
};
```

where:

- objectType—Refers to the type of object to be created. The complete list of object types and their corresponding `objectType` values are defined in the Attribute.idl file. For example, to create managed objects, this should be `ATL_OBJ::MANAGED_OBJECT`.
- classID—Specifies the object class to be created.
- containments—Specifies the containment relationships of the new object, encapsulated in a vector of ContainmentSpec data structures.

ContainmentSpec is defined as:

```
struct ContainmentSpec
{
    enum ContainmentStatus
```

```
    {
        Path_unprocessed,
        Path_okay,
        Path_invalidParent,
        Path_badTree,
        Path_error
    } status;
    ATL_OBJ::ObjectID treeID;
    ATL_OBJ::ObjectID parentID;
    string name;
};
```

- attributes—Specifies attributes which should be set when the object is created.
- participantAttributes—A list reserved for participants to add extra attributes to the object.
- name—Specifies the name of the subscriber object.
- return value of the method is an ATL_OBJ::ObjectID handle for the ObjectDefinition just added to the context.

As we have already seen, the object definitions are held within the DeploymentContext in Layers, each Layer corresponding to a Cisco EMF object type. All the ObjectDefinitions in a Layer can be retrieved for inspection. The interface also provides methods for getting and setting the properties of the object definitions.

Deployment Error Handling

As previously discussed, deployment can only have one of two outcomes, complete success or complete failure. If there is a failure to create something in the context, then the DeploymentContext will start to rewind and all that it had successfully completed up to that point will be undone. Rewind will go backwards through all of the levels visited so far, with an flag set to indicate the direction of processing:

```
module ATL_PART
{
    enum ParticipationDirection
    {
        Context_going_forward,
        Context_going_backward
    };
};
```

Each Participant can indicate if it is interested in rewind when it registers at a given level by use of this flag:

```
module ATL_PART
{
    enum ParticipationDirectionInterest
    {
        Interest_forward,
        Interest_backward,
        Interest_both,
        Interest_unknown
    };
};
```

With these semantics, either everything in the DeploymentContext gets created or nothing does. It implies multi-object rollback. Each ObjectDefinition in the context has a status value which, at the end of the process, will indicate whether or not it encountered an error condition during the creation process.

Participation Levels Used in Deployment

Each Participant registers with the ParticipationCoordinator at one or more deployment levels. In the majority of cases, Controller developers will only be interested in the Definition and Action levels for Managed Objects and Maps.

There are many deployment levels in Cisco EMF and any of them may be registered to. It is common that a participant may wish to register to more than one. For a full list of the levels refer to the Deployment.idl.

Consider, for example, a client who wishes to be involved in object creation, such that when an object is about to be created it may want to add sub-objects. In this case the client would register at the "Provision_Definition_MO_Expand" level, or in reality the number used to represent this level. The client would then be passed the context at the relevant part of a provisioning participation cycle, then it would take the context and augment it.

These levels and their numeric values are defined in the Deployment IDL file for users' convenience.

For more information regarding deployment, refer the *Cisco Element Management Framework SDK Controller Designer Developer's Guide.*

Naming and Identifying Cisco EMF Objects

Note Refer to Chapter 2, "Basic Concepts" for an overview of the concepts in Cisco EMF and applications, which should be understood prior to starting development of any Cisco EMF-based integration solution using the CORBA Gateway.

AtlNaming API

The AtlNaming IDL file defines the ATL_META module. The AtlNaming interface is used to translate between the names of Cisco EMF objects, and their ObjectIDs. The AtlNaming interface is defined as follows:

```
module ATL_META
{
    interface AtlNaming
    {
        short idToName(
             in ATL_OBJ::ObjectIDList oil,
             out ATL_OBJ::ObjectIDNamePairList successes,
             out ATL_OBJ::ObjectIDList failures);

        short nameToId(
             in ATL_OBJ::ObjectNameList names,
             in AtlType type,
             out ATL_OBJ::ObjectIDNamePairList successes,
             out ATL_OBJ::ObjectNameList failure);
    };
};
```

where `ATL_OBJ::ObjectIDList, ATL_OBJ::ObjectIDNamePairList and ATL_OBJ::ObjectNameList` are defined as follows:

```
module ATL_OBJ
{
    struct ObjectIDNamePair
    {
        ObjectID objectID;
        string objectName;
    }
    typedef sequence<ObjectIDNamePair>
    ObjectIDNamePairList;
    typedef sequence<string> ObjectNameList;
    typedef sequence<ObjectID> ObjectIDList;
};
```

Cisco EMF client applications typically make use of the nameToID() method to retrieve the ObjectIDs for containment trees, object classes, and attributes etc as part of their initialization phase, and then use the ObjectIDs as required in calls to the other servers.

The names to be looked up are submitted as an ObjectNameList, and the successfully resolved names returned as the parameter `successes`, which is a list of name/ObjectID pairs. Failures are returned as the list of names, `failures`. Only the names of objects of a single type can be submitted to nameToID() method at one time. The type is identified by the enumeration AtlType.

Table 6-1 Allowed Values for AtlType, and their Usage

Value of AtlType	Corresponding Object Type
CONTAINMENTOBJECTTYPE	Containment object
ALARMTYPE	Alarm object (reserved for future releases)
SECTIONTYPE	Section
ATTRIBUTETYPE	Attribute
CLASSTYPE	Managed object class
TYPETYPE	Managed object type
TREETYPE	Containment tree object
TYPEACTIONTYPE	Type Action object
ACTIONSCENARIOTYPE	Action Scenario object

Similarly, the method idToName() can be used to return the names of objects identified by their ObjectIds. In this case, the input data, and the returned failures are lists of ObjectIDs. There is no need to specify the object type in this case, since this is encoded in the ObjectID.

Note These methods are synchronous, and therefore it is not necessary to instantiate and listen for events on a callback object. It is also not necessary to supply a userData parameter.

Accessing and Modifying Attributes of Cisco EMF Objects

Fundamental to Cisco EMF is the abstract object model. Access to attributes on objects within this object model is a clear requirement, this requirement is satisfied by the CORBA DataAbstractor. The CORBA DataAbstractor interface provides access to values within Cisco EMF . It presents both a synchronous and asynchronous interface to enable the getting and setting of attribute values on objects within the system.

Examples of use for the CORBA DataAbstractor include the walking of Containment Trees, the setting of attributes on managed objects, for example setting the SNMP version of an agent, or the retrieval of attribute history data for polled objects.

DataAbstractor API

The DataAbstractor IDL file defines the ATL_DABS module. The DataAbstractor interface grants clients access to the data stored by Cisco EMF. The DataAbstractor interface, and the DataAbstractorCallback interface used to return the results of asynchronous queries, are defined in the following IDL:

```
module ATL_DABS
{
    interface DataAbstractorCallback;
    interface DataAbstractorCallbackExtended;

    interface DataAbstractor {

        short getAsync( in ATL_OBJ::ObjectAttributeList requests,
            in DataAbstractorCallback daCallback,
            in unsigned long userData );

        short setAsync( in ATL_OBJ::ObjectAttributeList requests,
            in DataAbstractorCallback daCallback,
            in unsigned long userData );

        void cancelAsync();

        short get(inout ATL_OBJ::ObjectAttributeList requests);

        short set(inout ATL_OBJ::ObjectAttributeList requests);

        short setAVVSimple ( in ATL_OBJ::AVVSimpleList requests,
            in DataAbstractorCallbackExtended daCallback,
            in unsigned long userData );
```

```
    };

interface DataAbstractorCallback {

    oneway void getCB( in ATL_OBJ::ObjectAttributeList results,
        in unsigned long userData );

    oneway void setCB( in ATL_OBJ::ObjectAttributeList results,
        in unsigned long userData );

};

interface DataAbstractorCallbackExtended :  DataAbstractorCallback {

    oneway void setAVVSimpleCB( in ATL_OBJ::AVVSimpleList success,
        in ATL_OBJ::AVVSimpleFailureList failures,
        in unsigned long userData );

    };

};
```

The data is passed both in and out of the server in the form of a ObjectAttributeList, defined in the ATL_OBJ module as shown below:

```
module ATL_OBJ
{
    struct ObjectAttributeListEntry
    {
        ObjectID id;
        AttributeList attrs;
        AttributeList fails;
    };
    typedef sequence<ObjectAttributeListEntry> ObjectAttributeList;
};
```

where `Attribute` is defined as:

```
module ATL_OBJ
{
    struct Attr
    {
        ObjectID attrID;
        AttributeValueUnion attrValue;
    }
};
```

and `AttributeValueUnion` is a union structure of the value of the attribute, discriminated by attribute type.

The setAVVSimple method call defines an interface for setting attributes of type VectorValue. VectorValue is used to represent SNMP table data.

```
module ATL_OBJ
{

    enum VectorValueStructureEnum
    {
        COMPLETE_INDEXED_TABLE,
        PARTIAL_INDEXED_TABLE,
        PARTIAL_INDEXED_TABLE_REMOVE
    };
```

```
enum VectorValueFillStateEnum
{
    COMPLETELY_FILLED,
    INCOMPLETELY_FILLED
};

enum EnumSNMPTableIterationStatus
{
    POSSIBLY_MORE_ROWS_TO_RETURN,
    NO_MORE_ROWS_TO_RETURN
};

enum SNMPStateFlagEnum
{
    SET,
    UNSET
};

struct SNMPSpecificStruct
{
    long                          rowsPerGet;
    EnumSNMPTableIterationStatus isComplete;
    SNMPStateFlagEnum             metadataSetStatus;
    AttributeList                 fixedIndexValues;
    SNMPStateFlagEnum             fixedIndicesSetStatus;
};

union SNMPSpecific switch (boolean)
{
    case TRUE :
        SNMPSpecificStruct snmpStruct;

};

struct VectorValue
{
    VectorValueStructureEnum valStruct;
    VectorValueFillStateEnum fillState;

    long    rows;
    long    cols;

    ObjectIDList   colIds;
    ObjectIDList   fixedIndices;
    ObjectIDList   fixedIndexMappings;
    ObjectIDList   freeIndices;
    AttributeList   storage;
    SNMPSpecific snmpSpecificOptional;

};

struct AVVSimple
{
    ObjectID moid;
    ObjectID attrid;
    VectorValue value;
};

typedef sequence<AVVSimple> AVVSimpleList;

struct AVVSimpleFailure
{
    ObjectID moid;
    ObjectID attrid;
```

```
    };

    typedef sequence<AVVSimpleFailure> AVVSimpleFailureList;

};
```

Asynchronous DataAbstractor API

The method getAsync() is used to access the current values of attributes. To fetch the values of attributes of a single Cisco EMF object, the ObjectIDs of the attributes are placed into the attrID fields of an attrs list of an ObjectAttributeListEntry. The fails list is left empty. The object is identified by its ObjectID in the id field. In this case, the ObjectAttributeList would only have one entry. The attributes of more than one object can be read in the same call by creating several entries in the ObjectAttributeList, one for each Cisco EMF object. The results of the call are returned to the client by the method `getCB()`. A new ObjectAttributeList structure is used, the attribute values will have been filled in the attrs list, and any attributes which could not be read will have been placed on the fails list.

The method `setAsync()` is used to set values of attributes. It is used in a very similar way to `getAsync()`, except that the desired new values of the attributes are filled in by the client.

The method cancelAsync cancels all outstanding asynchronous replies for that particular client's Data Abstractor object. Note this does not rollback (i.e. unset) outstanding set commands, doing a cancelAsync merely means that the client is not called again. This is useful when a client wishes to exit without waiting for all the outstanding requests to finish.

Synchronous DataAbstractor API

As well as the asynchronous methods getAsync and setAsync there are two synchronous methods, get and set. These synchronous methods provide the same functionality as the asynchronous methods and exist primarily to support client applications that do not wish to use an asynchronous design model or have other considerations. Prime examples of client applications where the use of get and set would be appropriate are:

- Threaded clients
- Clients where security make the use of the callback pattern difficult (e.g. Java applets)

CHAPTER 8

Launching Cisco EMF Element Manager Actions

ActionLauncher API

Actions are high-level commands which perform tasks on an EMS. These include the tasks which can be launched by clicking on a button on the EMS GUI, for example, connecting a new subscriber to a service. Actions can be scheduled, that is, given a specific execution time. Actions could be used to change the state of a managed network object, for example, to reset an object to clear an error, or to put an object into self-test mode.

When an action is complete, an action result is sent back to inform the initiator of the outcome.

The ActionLauncher IDL file defines the ATL_ACTION module. The ActionLauncher interface is defined as follows:

```
module ATL_ACTION
{
    struct Action
    {
        string actionName;
        unsigned long time;
    };

    struct ObjectTypeDescriptor
    {
        ATL_OBJ::GenericIDVectorValue objectIDs;
        ATL_OBJ::AttributeList attributes;
        string typeName;
    };

    typedef sequence<ObjectTypeDescriptor> ObjectTypeDescriptorList;

    struct ActionScenario
    {
        Action action;
        ObjectTypeDescriptorList otds;
        long userData;
    };

    typedef sequence<string> Annotations;

    struct ActionResult
    {
        boolean status;
        long userData;
        ATL_OBJ::ObjectAttributeList oal;
        Annotations annotations;
    };
```

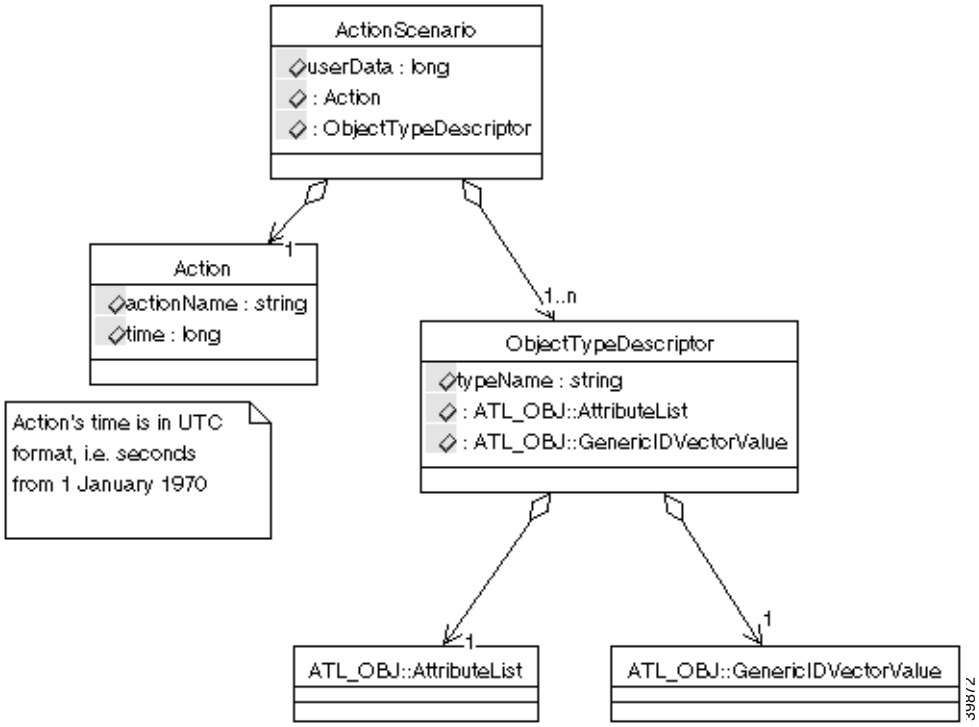
```
    interface ActionResultCallback;

    interface ActionLauncher {

    short invokeAction(in ActionScenario as,
        in ActionResultCallback cb,
        in unsigned long userData);

    };

    interface ActionResultCallback {

    void actionInvocationResult( in ActionResult result,
        in unsigned long userData);

    };

};
```

The structure ActionScenario (Figure 8-1) specifies all the details of the action to be carried out. The structure Action identifies the action itself by name, and also includes the time at which the action should be performed. The ObjectTypeDescriptorList's purpose is to hold the data for an action. The way in which this data is created and interpreted is solely a matter for a specific action's designer. Every action contains the time at which the action will be executed. This time is held in Universal Time Coordinated (UTC) format, which is the number of seconds that have elapsed since midnight on 1 January 1970.

The CORBA client invoking an action gets called back once the action has been handled, this is done via the ActionResultCallback interface. The client can use the status (succeed or fail), annotations (this may be text the CORBA client can echo back to the user) and an ObjectAttributeList in ActionResult structure to get the information on the action.

Note The annotations and ObjectAttributeList can be populated with data in a completely generic manner for the purpose of a specific action.

Figure 8-1 ActionScenario Structure



Event Channels

Event channels are used to distribute events in Cisco EMF both internally and between processes. The concept is simple. A user opens an event feed with a given filter on a given channel. When events are raised on that channel the users filter evaluates the events. Each event which passes the filter is delivered to the user. In this way event channels provide an efficient method of informing users of change.

The CORBA Event Channel Manager service provides clients with visibility of the Cisco EMF Event Channels and the ability to filter the events on these channels so only the ones of interest are received.

The main example of use for the CORBA Event Channel Manager is when the CORBA client and Cisco EMF need to maintain a consistent view of the object model. Consider a situation where a CORBA client is maintaining some representation of an Object Group, perhaps to display in a GUI. It is clear that this representation must remain consistent with the framework view of the group. Instead of continually having the CORBA client use the Object Group Server to iterate over the contents of the group to detect changes, which is very inefficient, the client can register for OGChangeEvents on the appropriate event channel and update as necessary.

Information in Events is completely generic, an event's content can be as varied as deployment information or SNMP trap data. CORBA clients have access to these events in a very dynamic and efficient manner. CORBA clients receive events via Iona's implementation of Notification Service (please refer to the Iona and OMG documentation in Appendix A, "Other Resources").

Cisco EMF events are first converted to CORBA equivalents. These are then pushed onto CORBA Event Channels within the Notification Service. The events are thereby broadcast to connected clients of the channels.

The way in which the channel clients (known as consumers) wish to receive events is dependent on the individual consumer. A consumer may choose between a push or pull model for receiving events. The push model calls the consumer asynchronously as soon as events are ready for it. If the pull model is used the channel stores events for it until it decides to call the channel to collect them.

An individual consumer is able to dictate the Quality of Service (QoS) it desires.

For efficiency consumers have the ability to turn off and on event types they will receive, this is to prevent the unnecessary creation of unwanted events. Switching off event types reduces the Notification Service processing and in turn can save Cisco EMF processing.

As part of the Notification Service the consumers are able to filter events on the data carried in what is known as the filterable_body of the Structured Events (for a complete description of Structured Events, refer to the Iona and OMG documentation in Appendix A, "Other Resources"). Although consumers may, if they wish, filter on any part of the data payload, known as the remainder_of_body.

Note remainder_of_body filtering should be unnecessary and carries with it performance penalties.

The Cisco EMF CORBA Event Channel Service has two methods:

- A consumer can request the names of all available channels
- Provides a means of retrieving a channel's reference

For the complete definition of the Notification Service's Channels, Structured Events, Admin, filtering and ancillary objects that will be referred to in this section, please refer to the Iona and OMG documentation in Appendix A, "Other Resources".

Overview of CORBA Channels and Cisco EMF Events

Cisco EMF has many different types of events and any of these may be broadcast on any interested channel. The CORBA Events Manager provides references to CORBA channels and allows you to retrieve the CORBA channels in order to transport the CORBA Events. The Structured Event type of CORBA Event is used for every event produced.

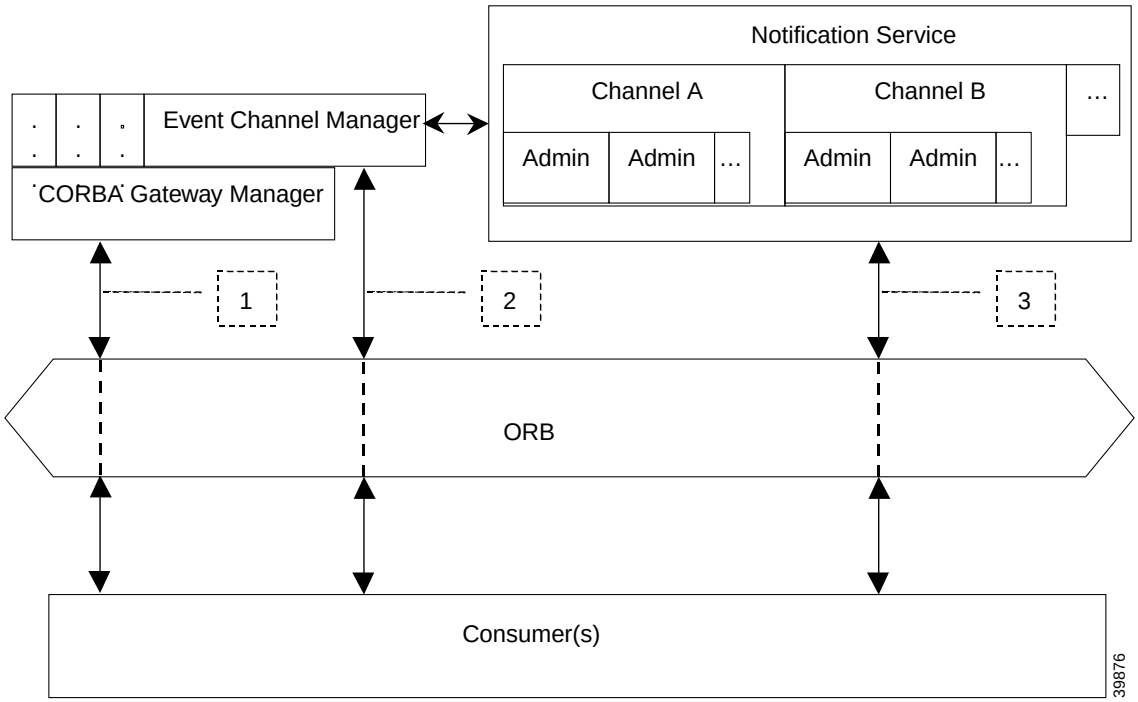
The following internal Cisco EMF events can be translated into a CORBA equivalent:

- OGCreateEvent
- OGDeleteEvent
- OGChangeEvent
- SnmpTrapEvent
- GenericObjectEvent
- TreeRenamedEvent
- TreeRemovedEvent
- TreeAddedEvent
- MetadataDistributionEvent
- ParenthoodRemovedEvent
- ParenthoodAddedEvent
- ObjectRenamedEvent
- ObjectRemovedEvent
- ObjectAddedEvent
- AlarmEvent

Architecture

Figure 9-1 shows the architecture of the CORBA Event Channels. Initially it is necessary to obtain a reference to the CORBA Gateway. Having this we can request the service called EventChannelManager. This service provides the consumer with a list of the names of the available channels. The EventChannelManager can translate the name of the channel into a channel reference for the consumer.

Figure 9-1 CORBA Event Channel Architecture



1 Get Event Channel Manager Service
2 Get Available Channels / Get Channel Reference
3 Add Filter / Push or Pull Events

The Notification Service provides multiple channels, each one can have multiple objects called CosEventChannelAdmin::ConsumerAdmin associated with them. The purpose of these objects is to group proxies to the consumers to allow the aggregation of filtering and Quality of Service settings.

The Notification Service's CosNotifyChannelAdmin module has an interface called the ProxySupplier which provides the consumer with the available event types via the obtain_offered_types method. The consumer having the results of this method is then free to add and remove the types it wishes to receive. This is done through the ProxySupplier method subscription_change.

Event Channel API

The ATL_EVENT module is defined in the EventChannelManager IDL file. The interface EventChannelManager allows clients to retrieve all the available channel names via the getAvailableChannels method, and more importantly, it allows them to obtain a reference to any of the channels by passing the channel name to the getChannel method.

```
module ATL_EVENT
{
        interface EventChannelManager;

    interface EventChannelManager{

        exception NoSuchEventChannel{
    };

    /**
        */
        typedef sequence<string> ChannelNameList;

        /**
        */
        ChannelNameList getAvailableChannels();

        CosNotifyChannelAdmin::EventChannel getChannel(in string channelName)
            raises(NoSuchEventChannel);
    };
};
```

The structure of the CORBA Events are defined in the StructuredEvent IDL file. The ATL_STRUCTURED_EVENT module in this file defines the structure of Cisco EMF's CORBA events.

As an example we can look at the OGChangeEvent. The comments here under the label "Filterable" clearly state the data that is to be provided in the Structured Event's filterable_body . The "struct" definition is the complete set of data relating to the event. This struct is inserted into the Any of the Structured Event's section called remainder_of_body. This data can be regarded as the event's payload.

The online documentation provided with the Developer's Toolkit clearly lists all the various fields.

```
    /**************************************************************************
    OGChangeEvent Provides information on the changes to an object group
    Filterable:-
        @member groupName Actual group name
        @member groupId Id of the group the changes refer to
        @member groupType Defines the persistence etc
Non-Filterable:-
        @member added ids of the object added
        @member changed
        @member removed object removed from object
        @member oalData data in OAL form
*/
struct OGChangeEvent
{
        string  groupName;
        ATL_OBJ::ObjectID groupId;
        ATL_OG::GroupType groupType;
        ATL_OBJ::ObjectIDList added;
        ATL_OBJ::ObjectIDList changed;
        ATL_OBJ::ObjectIDList removed;
        ATL_OBJ::ObjectAttributeList oalData;
    ;
```

Creating a Consumer

Using code fragments as examples this section will show the necessary steps to creating a consumer. This consumer will connect to the OGChannel and receive event types OGCreateEvent and OGDeleteEvent. These are as follows:-

- Use the CORBA Gateway to access the EventChannelManager Service
- Implementing a Consumer Interface
- Get a Channel Reference from the EventChannelManager
- Subscription to Event Types
- Add Filter
- Start receiving events

Use the CORBA Gateway to access the EventChannelManager Service

Assuming that `corbaGatewayManager_var` is an `ATL_GATE::CorbaGatewayManager_var` and correctly references the CORBA Gateway. We access the EventChannelManager service as follows:

```
    //
    // Get Object From Gateway
    //
    CORBA::Object_ptr objFromGateway = CORBA::Object::_nil();
    try
    {
        objFromGateway = corbaGatewayManager_var->get("EventChannelManager",
            user,
            password);
    }
    catch(ATL_EXCEPT::invalidServerName invalid)
    {
        /* do appropriate action for exception */
    }
    catch(ATL_EXCEPT::authorisationFailure author)
    {
        /* do appropriate action for exception */
    }

    if( CORBA::is_nil(objFromGateway) )
    {
        /* do appropriate action for error */
    }

ATL_EVENT::EventChannelManager_var eventManager_var;

    eventManager_var = ATL_EVENT::EventChannelManager::_narrow(objFromGateway);
```

Implementing a Consumer Interface

It is up to the client to decide which of the CosNotifyComm module consumer interfaces it wishes to implement. Here for example is a consumer which implements the SequencePushConsumer interface and will therefore be pushed events in batches. For all implementation the consumer is notified of changes to event types available via the offer_change method.

```
#include <EventChannelManager.hh>
#include <CosNotifyComm.hh>

class TestConsumer : public CosNotifyComm::SequencePushConsumerBOAImpl
{
    public:
    TestConsumer();

    /**
        Receive events from the channel
    */
        void push_structured_events(
            const CosNotification::_IDL_SEQUENCE_CosNotification_StructuredEvent& events,
            CORBA::Environment &IT_env = CORBA::IT_chooseDefaultEnv() )
            throw (CORBA::SystemException, CosEventComm::Disconnected);

        virtual void disconnect_sequence_push_consumer(
            CORBA::Environment &IT_env = CORBA::IT_chooseDefaultEnv () )
            throw (CORBA::SystemException)
        {
        }

        /**
             Callback from admin to issue new event types
        */
        virtual void offer_change (const CosNotification::EventTypeSeq& added,
            const CosNotification::EventTypeSeq& removed,
            CORBA::Environment &IT_env =
            CORBA::IT_chooseDefaultEnv ());

    private:
};
```

Get a Channel Reference from the EventChannelManager

Step 1 If we wish to obtain a channel reference, we should first check to see which channels are available. The following code example does this. It displays the number of available channels and their names on STDOUT

```
ATL_EVENT::EventChannelManager::ChannelNameList *names;
try
{
    names =
        m_eventManager->getAvailableChannels();
}
catch(CORBA::Exception &e)
{
    /* do appropriate action for exception */
}

const int length = names->length();
int i= 0;
cerr << "There are " << length << " channels available...." << endl;

for(;i < length; i++)
{
    cerr << CORBA::string_dup((*names)[i]) << endl;
    }
```

Step 2 Having found the channel we are looking for, we obtain a reference to the CosNotifyChannelAdmin::EventChannel. Here we obtain a channel reference for the OGChannel channel as follows.

```
const char *channelName = "OGChannel";
try
{
    chan = m_eventManager->getChannel(channelName);
}
catch(ATL_EVENT::EventChannelManager::NoSuchEventChannel)
{
    /* do appropriate action for exception */
}
catch(CORBA::Exception &e)
{
    /* do appropriate action for exception */
}
```

Step 3 Next, get a reference to a CosNotifyChannelAdmin::ConsumerAdmin object from the channel. An Admin object allows the setting of QoS parameters and filtering for a set of proxies. We have the choice of getting the default admin object for the channel via the call where zero is passed as the id:

```
get_consumeradmin(in AdminID id)
```

or if we wish a new Admin object we use the call

```
ConsumerAdmin new_for_consumers(in InterFilterGroupOperator op, out AdminID id);
```

Step 4 Once we have the Admin we can create a proxy to the channel's supplier(s) as the following code fragment shows:

```
CosNotifyChannelAdmin::SequenceProxyPushSupplier_var seqPPS;
    CosNotifyChannelAdmin::ConsumerAdmin_var admin;
    CORBA::Object_var tmpObj = CORBA::Object::_nil();
    try
    {
        admin = chan->get_consumeradmin(0);

        tmpObj = admin->obtain_notification_push_supplier(
            CosNotifyChannelAdmin::SEQUENCE_EVENT , pid);
    }
    catch(CORBA::Exception &e)
    {
        /* do appropriate action for exception */
    }

    seqPPS = CosNotifyChannelAdmin::SequenceProxyPushSupplier::_narrow(tmpObj);
```

Subscription to Event Types

The consumer is required to choose which event types it wishes to receive. It can determine the event types the channel distributes via the CosNotifyChannelAdmin::ProxySupplier interface's method

```
CosNotification::EventTypeSeq obtain_offered_types();
```

To add or remove subscriptions from event types the consumer uses the subscription_change method in the CosNotifyComm module's NotifySubscribe interface. In the following code fragment assume that seqPPS is a reference to a Sequence Proxy Supplier retrieved in the previous example:

```
CosNotification::EventTypeSeq_var add(2);
CosNotification::EventTypeSeq_var remove;
add->length(2);

(*add)[0].domain_name = CORBA::string_dup("Telecommunications");
(*add)[0].type_name = CORBA::string_dup("OGCreateEvent");

exp[0].constraint_expr =

CORBA::string_dup("$.filterable_data(groupName)=='OG_6400'");

(*add)[1].domain_name = CORBA::string_dup("Telecommunications");
(*add)[1].type_name = CORBA::string_dup("OGDeleteEvent");

try
{
    seqPPS->subscription_change( add, remove);
}
catch(CosNotifyComm::InvalidEventType &iet)
{
    /* do appropriate action for exception */
}
catch(CORBA::Exception &e)
{
    /* do appropriate action for exception */
}
```

Add Filter

The details of filtering are fully explained in the Iona and OMG documentation, refer to Appendix A, "Other Resources". Here is an example of a filter which will only allow an event of type OGDeleteEvent if its groupId matches.

```
CosNotifyFilter::FilterFactory_var filterFac = chan->default_filter_factory();
CosNotifyFilter::Filter_var filter = filterFac->create_filter("EXTENDED_TCL");
CosNotifyFilter::ConstraintExpSeq exp = CosNotifyFilter::ConstraintExpSeq(1);
exp.length(1);
exp[0].event_types = CosNotification::EventTypeSeq(1);
exp[0].event_types.length(1);
exp[0].event_types[0].domain_name = CORBA::string_dup("Telecommunications");
exp[0].event_types[0].type_name = CORBA::string_dup("OGDeleteEvent");
exp[0].constraint_expr=  CORBA::string_dup(
    " $groupId.upper == 0xa && $groupId.lower == 0x1a34");

CosNotifyFilter::ConstraintInfoSeq_var cis;
try
{
    cis1 = filter->add_constraints(exp);
}
catch(CosNotifyFilter::InvalidConstraint& _exobj1)
{
    /* do appropriate action for exception */
}

CosNotifyFilter::FilterID id1 =  seqPPS->add_filter(filter);
```

Note A filter may be added to a channel's admin object in the same manner as above. Doing this will apply the filter apply to the collection of proxies managed by the admin object.

Start receiving events

To receive the events the consumer must have already registered with the Implementation Repository. To wait for incoming calls is necessary to call `impl_is_ready`. If the consumer wishes to disconnect from the channel it is important that it removes its subscriptions and then disconnects as shown below.

```
const int timeoutMilliSec = 60000;
const char *serverName = "ChannelConsumer";
try
{
    CORBA::Orbix.impl_is_ready(servername,timeoutMilliSec);
}
catch(CORBA::Exception &e)
{
        /* do appropriate action for exception */
}

/* Remove all subscriptions */
CosNotification::EventTypeSeq_var add;
CosNotification::EventTypeSeq_var remove(2);
add->length(2);

(*remove)[0].domain_name = CORBA::string_dup("Telecommunications");
(*remove)[0].type_name = CORBA::string_dup("OGCreateEvent");

(*remove)[1].domain_name = CORBA::string_dup("Telecommunications");
(*remove)[1].type_name = CORBA::string_dup("OGDeleteEvent");

try
{
    sequence_pps->subscription_change( add, remove);
}
catch(CosNotifyComm::InvalidEventType iet)
{
    /* do appropriate action for exception */
}
catch(CORBA::Exception &e)
{
    /* do appropriate action for exception */
}

m_sequence_pps->disconnect_sequence_push_supplier();
```

CHAPTER 10

Object Groups

The CORBA Object Groups API allows clients to manipulate Cisco EMF Object Groups.

An object group is simply a collection of objects which are related in some way. They may all be the same type of equipment or all belong to the same customer.

Object groups can be built either manually or by building a query. Some Cisco EMF subsystems may also build object groups which may be visible and usable by the Cisco EMF user.

Further information on Object Groups may be found in the *Cisco EMF Developer Concepts Manual*.

Object Groups API

The ATL_OG module is defined in the ObjectGroups IDL file and encompasses seven different interfaces:

- OGManager—Provides methods for gaining references to ObjectGroup interfaces. This interface also allows CORBA clients to move object ids from one object group to another, via the moveObjects method.
- ObjectGroup—Provides functionality for retrieving the details of an object group, for instance its name or cardinality, and also the operations that can be performed on an object group like renaming or copying. The ObjectGroup interface also provides functions for retrieving an ObjectGroupIterator interface which allows the retrieval of group contents.
  - ManualObjectGroup—Derives from ObjectGroup. Specialized for Manually Populated Object Groups.
  - QueryObjectGroup—Derives from ObjectGroup. Specialized for Query Populated Object Groups.
- ObjectGroupIterator—Gives access to general Iterator attributes such as name and cardinality. ObjectGroupIterator has two derived interfaces, ObjectGroupSimpleIterator which simply retrieves the object ids from the group and ObjectGroupOperationIterator which retrieves the objects in the group together with the per object result of an attribute set or get operation.
  - ObjectGroupSimpleIterator—Derives from ObjectGroup. Specialized for Simple Object Group Iteration.
  - ObjectGroupOperationIterator—Derives from ObjectGroup. Specialized for Object Group Iteration with an Operation.

Creating an Object Group

The following example demonstrates the creation of an Object Group using the CORBA Gateway.

The steps which are involved in this process are as follows:

- Use the CORBA Gateway to Access the ObjectGroups Service
- Use the Participation Interface to Create a New Object Group
- Use OGManager to Obtain Reference to the New Object Group

Use the CORBA Gateway to Access the ObjectGroups Service

The following code sample shows how the service can be obtained assuming that `corbaGatewayManager_var` is a previously obtained valid reference to the CORBA Gateway Manager:

```
//
// request an object from the CORBA Gateway Manager
//
    CORBA::Object_var tempObj;
try
{
            tempObj    = corbaGatewayManager_var->get(
        "ObjectGroups",
        user,
            password);
}
    catch(ATL_EXCEPT::invalidServerName invalid)
    {
        // do appropriate action for exception
    }
    catch(ATL_EXCEPT::authorisationFailure author)
    {
        // do appropriate action for exception
    }

ATL_OG::OGManager_var ogManager_var;

//
// narrow the object to the required type
//
ogManager_var = ATL_OG::OGManager::_narrow(tempObj);

    if (CORBA::is_nil(ogManager_var))
{
    // deal with the error
    }
```

Use the Participation Interface to Create a New Object Group

The following code example demonstrates the necessary stages for the creation of a new Object Group. We will assume that references to the AtlNaming, ParticipationCoordinator and DeploymentContext interfaces have been successfully obtained and narrowed to the correct type. Further information regarding the AtlNaming interface can be found in the relevent chapter and the accompanying HTML documantation.

The following example will create a Persistent, Query Populated Object Group with name Example Group and description Example Object Group Creation.

Firstly, it is necessary to store all the details of the group we wish to create inside attributes which can then be passed to the Deployment Context.

```
//
// the attributes containing creation information for the OG
//
    ATL_OBJ::AttributeList attrList;
    ATL_OBJ::Attr oGTypeAttr, queryAttr, descAttr;

//
// this is the type of Object Group that we are deploying
//
    oGTypeAttr.attrID.upper = ATL_SystemAttributes::attributeUpper;
    oGTypeAttr.attrID.lower = ATL_SystemAttributes::ogType;
oGTypeAttr.attrValue.int32Value(int(ATL_OG::PersistentManual));

//
// this attr holds the query that we will use to create an auto-populating OG.
// if we wish to create a manual OG then this attribute is unnecessary.
//
    queryAttr.attrID.upper = ATL_SystemAttributes::attributeUpper;
    queryAttr.attrID.lower = ATL_SystemAttributes::ogQuery; // construct the query later

//
// this is a textual description of the OG we are creating
//
    descAttr.attrID.upper = ATL_SystemAttributes::attributeUpper;
    descAttr.attrID.lower = ATL_SystemAttributes::ogDescription;
descAttr.attrValue.stringValue(CORBA::string_dup("Example Object Group Creation"));
```

The following code fragment constructs a query that will be used to populate our new Object Group.

```
//
// First we will create the Query
//

// the Containment Scope
ATL_Query::ContainmentScope contScope;
contScope.treeId = physicalId; // type ATL_OBJ::ObjectID
contScope.startId = start; // type ATL_OBJ::ObjectID
contScope.direction = 1;
contScope.startLevel = 1;
contScope.endLevel = 5;
contScope.includeStartObject = 1;
contScope.includeRootId = 0;

// the Object Type Filter
ATL_Query::ObjectTypeFilter typeFilter;
typeFilter.typeId = type; // ATL_OBJ::ObjectID

// the Query Spec List
ATL_Query::QuerySpecList list;
```

```
list.length(1);
list[0].scope.cs(contScope);
list[0].filter.otf(typeFilter);

// the entire Query
ATL_Query::Query query;
query.querySpecs = list;

//
// Now we can store this query inside the attribute, note that the value must
// be of type CORBA::Any
//
CORBA::Any anyQuery;
    anyQuery <<= query;
queryAttr.attrValue.queryValue(anyQuery);
```

Further information regarding the construction of Queries and the variety of filters that may be used can be found in the HTML documentation of the ATL_Query module.

We are now ready to add this object definition to the Deployment Context:

```
//
// put our creation attributes onto the list
//
    attrList.length(3);
        attrList[0] = oGTypeAttr;
        attrList[1] = descAttr;
        attrList[2] = queryAttr;

//
// add the object definition to the deployment context
//

// the type of object that is being deployed
//
CORBA::Octet objectType = ATL_OBJ::SYSTEM_PERSISTENT_OG;

// the name of our new object
//
CORBA::string oGName = CORBA::string_dup("Example Group");

deploymentContext_var->addObjectDefinition(objectType,
        ATL_OBJ::ObjectId oGClassId, // the class of our object.
        ATL_Deploy::ContainmentSpecVector(), // the containment of our object.
        attrList, // our list of creation attributes
        ATL_OBJ::AttributeList(), // any participant attributes for our deployment
        oGName);
```

Finally, we can make a call to the ParticipationCoordinator interface to begin the deployment. Assume 'part' is a valid pointer to a previously obtained ParticipationCoordinator object.

```
part->startContext(context,
        0,
        ATL_Deploy::Provision_Level_Base,
        ATL_Deploy::Provision_Level_End,
        this,
        0);
```

Note The call to `startContext` is asynchronous and the client will have to implement the necessary callback methods. Further information can be found in the chapter concerning the Participation interface and the HTML documentation.

Use OGManager to Obtain Reference to the New Object Group

Once the deployment context for the new group is complete we can use the ATL_OG::OGManager interface to obtain a reference to it.

```
// this list holds the names of the groups we wish to retrieve
//
ATL_OG::GroupNameList names(1);
names[0] = CORBA::string_dup(“Example Group”);

// this list will hold the obtained references
//
ATL_OG::ObjectGroupList_var groups = 0;

// this list will hold the names of any failures
//
ATL_OG::GroupNameList_var fails = 0;

if(!ogManager_var->getGroupsByName(names, groups, fails))
{
    // our get failed
}

// extract the reference from the results
//
ATL_OG::ObjectGroup_var = (*groups)[0];
```

Now that a valid reference has been obtained we can invoke the methods of the ObjectGroup interface.

Using the Simple Iterator

The ATL_OG::ObjectGroupSimpleIterator provides the client with the ability to easily recover the contents of an object group. Assuming a valid reference to an ATL_OG::ObjectGroup interface is held the simple iterator can be obtained as follows:

```
//
// obtain reference to ObjectGroupSimpleIterator
//
ATL_OG::ObjectGroupSimpleIterator_var iterator_var;

try
{
       iterator_var = objectGroup_var->simpleIterator();
}
catch(ATL_OG::OGGeneralFailureException &e)
{
    // deal with the exception
}
catch(CORBA::OGNotFoundException &e)
    {
        // deal with the exception
    }
catch(CORBA::Exception &e)
    {
        // deal with the exception
    }
```

After the reference to the iterator has been obtained the contents of the group can be retrieved by calling the following method defined in the ATL_OG::ObjectGroupSimpleIterator interface:

```
        /** Return the next portion of objects ids from the iterator. Note
        that the implementation controls how many elements are returned
        to the user, using the input parameter as a limiter.

        @param maxReturns the maximum number of elements to return
        @param elements the returned iterator elements

        @return true if operation succeeded, false otherwise
    */
boolean nextElements(
        in unsigned long maxReturns,
        out ATL_OBJ::ObjectIDList elements);
```

The following example demonstrates how to use this method to obtain group contents:

```
//
// iterate over group contents
//
CORBA::Boolean result;
ATL_OBJ::ObjectIDList_var objIdList_var;

result = iterator_var->nextElements(20, objIdList_var);

if(result == CORBA::FALSE)
{
    // deal with the failure
}
else // operation succeeded
{
    // process the object id list of results
}
```

Note It is the server that dictates the precise number of elements returned by the iterator, using the user supplied maximum as a limit.

The iterator can be reset back to the first element by calling the following method defined in interface ATL_OG::ObjectGroupIterator:

```
        /** Reset this iterator to start of iteration.
        */
        void reset();
```

It is important to understand that when a reference is obtained to an iterator interface it encompasses a snapshot of the contents of the group at that particular time. Subsequent changes to the group will not be reflected in the iterator. To repopulate the iterator the correct procedure is to first delete the reference to the current iterator interface (i.e. by calling `shutdown()`) and then retrieve a new reference to the iterator through the ObjectGroup interface.

Using the Operation Iterator

The example in this section will show how to use the ATL_OG::ObjectGroupOperationIterator interface. This interface, like the ObjectGroupSimpleIterator, is obtained through the ATL_OG::ObjectGroup interface. The method used to obtain the `ObjectGroupOperationIterator` is defined as follows:

```
/** Return an iterator over this object group. This form of
    iterator allows more complex iteration, including setting or
    retrieving attribute values on a per-object basis.
*/
ObjectGroupOperationIterator operationIterator(
    in GroupOperationType type,
    in ATL_OBJ::AttributeList attrs) raises(OGGeneralFailureException,
                                            OGNotFoundException);
```

The `type` parameter is an enumeration which specifies whether we wish to set the values of attributes or get the values. The `attrs` parameter is a list of attributes and their values. If we are performing a set operation then the new value for each attribute must be specified. However, in the case of an attribute get operation the value field for each attribute must be initialized to a null value, this is done as follows:

```
//
// setting an attribute value to null for an attribute get operation
//
ATL_OBJ::AttributeList attrsList;

attrsList.length(1);
attrsList[0].attrID = attributeId; // the id of some attribute
attrsList[0].attrValue.nullValue(1); // initialise the value to null
```

Further information on AttributeLists can be found in the ATL_OBJ module HTML documentation file.

The following example demonstrates how the use the ObjectGroupOperationIterator to set the values of two attributes on the contents of an object group and then to retrieve the results:

```
//
// first step is to construct an ATL_OBJ::AttributeList
//
ATL_OBJ::AttributeList_var attrList_var;

ATL_OBJ::ObjectID commentId;
ATL_OBJ::ObjectID readCommId;
/* initialise the above Ids with the ids of attributes
    LocalDB:AMAF-MGMT-MIB.Comment and LocalDB:SNMP-ATTRIBUTES-MIB.snmpv2c-read-community
    respectively. This could be done, for example, with the ATL_META interface.*/

attrList_var->length(2);
(*attrList_var)[0]->attrID = commentId;
(*attrList_var)[0]->attrValue.stringValue("A comment to set");
(*attrList_var)[1]->attrID = readCommId;
(*attrList_var)[1]->attrValue.stringValue("public");

//
// We can now go ahead and obtain the OperationIterator interface
//
ATL_OG::ObjectGroupOperationIterator iterator_var;

try
{
    iterator_var = objectGroup_var->operationIterator(ATL_OG::SetAttrs, attrList_var);
}
    catch(ATL_OG::OGGeneralFailureException &e)
{
    // deal with the exception
```

```
}
catch(ATL_OG::OGNotFoundException)
{
   // deal with the exception
}
catch(CORBA::Exception &e)
    {
        // deal with the exception
    }
```

Once we have a valid reference to the iterator we can retrieve the group contents and the operation results through a call to the nextElements() method. The method takes two parameters, the maximum number of iterator elements that the client wishes to receive and, in the case of the Operation Iterator, an ATL_OBJ::ObjectAttributeList structure that will hold the results. Further information regarding the `ObjectAttributeList` can be found in the ATL_OBJ module HTML documentation.

```
//
// get elements from iterator
//
ATL_OBJ::ObjectAttributeList oal_var;

try
{
    if(iterator_var->nextElements(20, oal_var) == CORBA::FALSE)
        // deal with the failure
}
    catch(ATL_OG::GroupDoesNotExist &e)
{
    // deal with the exception
}
catch(CORBA::Exception &e)
    {
        // deal with the exception
    }

// retrieve the results
int resultsLength = oal_var->results.length();

int current;
for(current = 0; current < resultsLength; current++)
{
    // process result
}
```

Note It is the server that dictates the precise number of elements returned by the iterator, using the user supplied maximum as a limit.

Calling Shutdown()

Once a reference to any of the ATL_OG interfaces is no longer needed by a client it is recommended that its `shutdown()` method is called. This lets the server know that the client no longer requires the reference, thus allowing it to remove the object and clean up. It should be stressed that it is only the CORBA servant objects that are deleted, there is no effect on the actual Cisco EMF Object Groups. The references used in the previous code fragment would be removed as follows:

```
//
// we no longer need these references
//
try
{
    ogManager_var->shutdown();
    iterator_var->shutdown();
}
catch(CORBA::Exception &e)
    {
        // deal with the exception
    }
```

Once shutdown has been called on an object reference it is important that the client invokes no further operations on it. Doing so will result in a CORBA exception being raised.

Exceptions

The module ATL_OG defines several exceptions the meanings of which are listed below :

- OGGeneralFailureException—The operation failed for an unspecified reason. No further information available. The groupId member of this exception contains the id of the group that the failed operation was called on.
- OGNotFoundException—All of the operations on ObjectGroup and ObjectGroupIterator interfaces (and their derived interfaces) throw this exception when the actual object group that the interface is associated with has been removed. This situation can occur if, for example, a client holds a reference to an ObjectGroup interface but the associated group has been removed. All calls to methods of these interfaces should be enclosed in a try-catch block which catches this exception, as well as the usual CORBA System Exceptions. The groupId member of this exception contains the id of the group that the failed operation was called on.
- OGGeneralMoveFailureException—This exception can only be raised by the OGManager::moveObjects method. It means that either the objects to be moved were not present in the source group of that they already exist in the destination group. This exception is only raised on complete failure.
- OGAlreadyExistsException—This exception can only be raised by the ObjectGroup::copy method. It means that the name specified for the copy destination already exists as an object group.

Example: Provisioning Client Written in C++

This real-life example of using the Cisco EMF CORBA Gateway will use the deployment scenario of connecting a new subscriber to a service. This procedure will illustrate the use of all the CORBA Gateway servers described in the preceding sections.

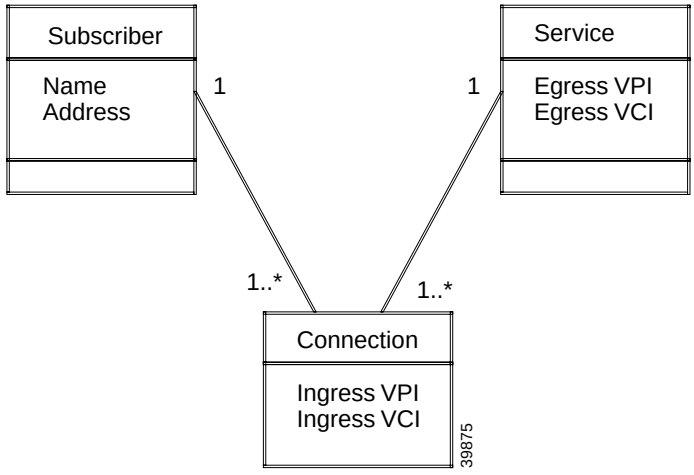
To achieve this, the client application will:

- Initialize the CORBA Gateway
- Create a new subscriber object in the Cisco EMF system (using Participation/Deployment)
- Locate a service object in the Cisco EMF system (using AtlNaming)
- Connect the subscriber to the service object (using the ActionLauncher)
- Check and amend an attribute of the new subscriber (using the DataAbstractor)

Element Manager Object Model

This example assumes that an EMS has been developed to manage an ATM network technology. Part of the object model for this EMS is shown in Figure 11-1.

Figure 11-1 : Example Element Manager Object Mode

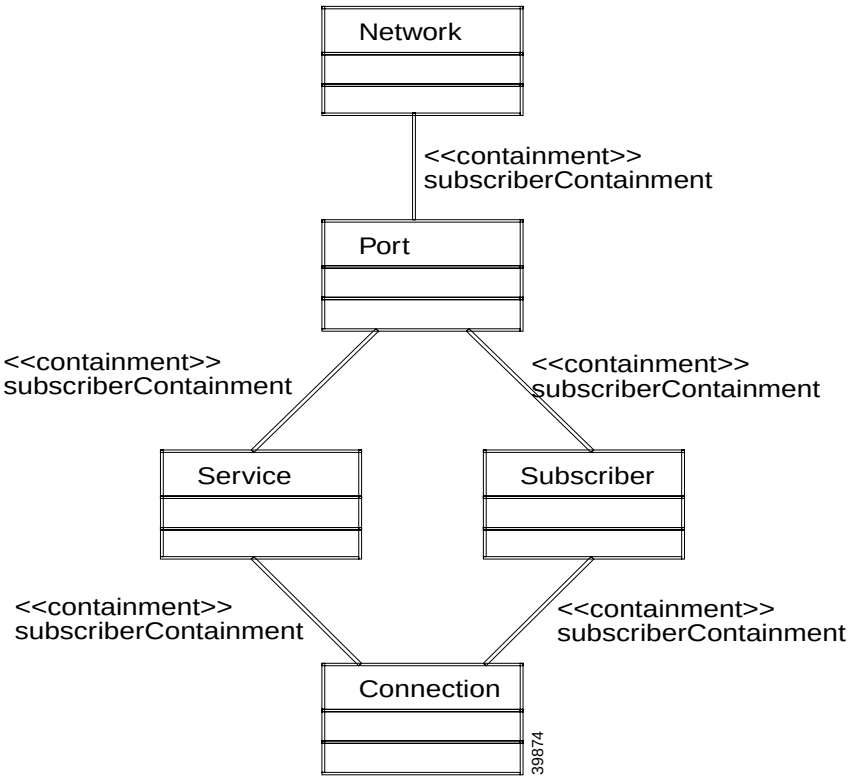


This part of the object model consists of three object classes:

- Service class—Represents a service being provided by a service provider which is connected to the ATM network being managed on a given port, with a specified VPI/VCI. The details of where the service is connected are stored as attributes of the Service object. For the purposes of this example, it will be assumed that the network administrator will manually create Service objects as new service providers are connected to his network using the GUI.
- Subscriber class—Represents a subscriber who is connected to the ATM network being managed on a given port. The name and address of the subscriber, are stored as attributes of the subscriber object. For the purposes of this example, it will be assumed that the Cisco EMF CORBA Gateway will be used to create subscriber objects using the deployment interfaces as part of a flow through provisioning integration. This will take orders for the provision of service to subscribers from an order handling system. Each order will contain details of the subscriber to be connected (including the name, and address values).
- Connection class—Represents an ATM PVC connecting a subscriber to a service. The VPI/VCI for this connection at the subscriber end are stored as attributes of the connection object. For the purposes of this example, it will be assumed that the Cisco EMF CORBA Gateway will be used to create connection objects using the action interfaces as part of a flow through provisioning integration. This will take orders for the provision of service to subscribers from an order handling system. Each order will contain details of the connections to be made (including the Ingress VPI and VCI values).

Instances of these classes live in a containment tree called subscriberContainment, with a structure as shown in Figure 11-2.

Figure 11-2 : Containment Relationships Between Objects



CORBA Client Startup Tasks

Clients written to interact with Cisco EMF using the CORBA Gateway will typically perform a number of tasks when they are started to prepare for use of the gateway interfaces. In the case of this example, the startup tasks are described in this section.

The Cisco EMF CORBA Gateway uses a service oriented strategy for its API. This means that the IDL includes interfaces to a small number of objects which allow services to be invoked, and does not use a CORBA object for each managed object. A CORBA client needs to retrieve object references for these objects as part of its bootstrap process. This process is handled by the CorbaGatewayManager interface, which provides a method to allow a client to obtain an object reference for the various server objects.

The first step, however, is for the client to initialize its ORB implementation (this example uses Orbix) and retrieve an object reference for the CorbaGatewayManager object itself by using the CORBA Naming Service to look it up by name.

The CORBA Client startup tasks include:

CORBA Initialization

Before any CORBA calls can be made, we must initialize the ORB. This is the C++ code fragment to initialize an Orbix client:

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "Orbix");
```

Get the CorbaGatewayManager Object Reference from the CORBA Naming Service

Resolve the Orbix Naming Service root context:

```
CORBA::Object_var ns =
    orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var nsRootContext =
    CosNaming::NamingContext::_narrow(ns);
```

Then obtain a reference to the Cisco EMF CORBAGatewayManager object by interrogating the CORBA Naming Service:

```
ATL_GATE::CorbaGatewayManager_var corbaGatewayManager_var;

CosNaming::Name_var name = new CosNaming::Name(2);
name->length(2);
name[0].id = CORBA::string_dup("atl");
name[0].kind = CORBA::string_dup("");
name[1].id = CORBA::string_dup("CorbaGatewayManager");
```

```
name[1].kind = CORBA::string_dup("ServerObject");

CORBA::Object_var tempObj = nsrc->resolve(name);
corbaGatewayManager_var =
    ATL_GATE::CorbaGatewayManager::_narrow(tempObj);
```

Note The **resolve_initial_references** call will only work for objects implemented in the same ORB environment, in this case, Orbix. For clients written using other ORBs, the Orbix Naming Service root context has to be resolved another way, by use of stringified IORs. Refer to Stringified Interoperable Object References, page 3-9.

Get AtlNaming Object Reference from CorbaGatewayManager

The AtlNaming interface allows clients to convert between names and the opaque identifiers used by the CORBA Gateway in many of its interfaces. The use of this interface is covered in Get ObjectIDs for Class, Attribute and Containment Tree Names from AtlNaming, page 11-6. A client will typically retrieve an object reference to an AtlNaming object in order to do name conversions as part of its bootstrap process. The AtlNaming object is obtained by using the get method of the CorbaGatewayManager interface, passing in "`AtlNaming`" as the serverName parameter:

```
Object get ( in string serverName,
    in string user,
    in string password)

    raises (ATL_EXCEPT::invalidServerName,
        ATL_EXCEPT::authorisationFailure);
```

This method instantiates an AtlNaming object and returns a reference to it. The server requested is identified by the serverName parameter, if the server name supplied does not correspond to a valid server, the invalidServerName exception is thrown. The user and password parameters are used to identify which Cisco EMF user the CORBA client is acting on behalf of. If these are not valid, the authorisationFailure exception is thrown. Here is a C++ code fragment to illustrate this:

```
CORBA::Object_var anObj =
    gateway_var->get("AtlNaming",
        "admin", "admin");
```

This call will throw an exception if an invalid server name is given, or if the user and password are not authorized. By analogy with the Naming Service, the returned object reference must be narrowed to the correct type, in this case `ATL_META::AtlNaming`:

```
ATL_META::AtlNaming_var atlNaming_var =
    ATL_META::AtlNaming::_narrow(anObj);
```

Get ParticipationCoordinator Object Reference from CorbaGatewayManager

The ParticipationCoordinator interface is used to support the deployment of managed objects on the Cisco EMF system. The use of this interface is covered in Creating the Subscriber Object Using CORBA, page 11-7. A client will typically retrieve an object reference to a ParticipationCoordinator object as part of its bootstrap process. The ParticipationCoordinator object is obtained by using the get method of the CorbaGatewayManager interface, passing in `"ParticipationCoordinator"` as the server name parameter. This method instantiates an ParticipationCoordinator object and returns a reference to it. Here is a C++ code fragment to illustrate this:

```
CORBA::Object_var pcObj =
    gateway_var->get("ParticipationCoordinator",
        "admin", "admin");
```

This call will throw an exception if an invalid server name is given, or the user and password are not authorized. By analogy with the Naming Service, the returned object reference must be narrowed to the correct type, which in this case is `ATL_PART::ParticipationCoordinator`:

```
ATL_PART::ParticipationCoordinator_var pCoord_var =
    ATL_PART::ParticipationCoordinator::_narrow(pcObj);
```

Get DeploymentContext Object Reference from CorbaGatewayManager

We need to obtain a reference to a DeploymentContext object that we will use to describe the subscriber object to be created. This interface is used in the steps Adding the Subscriber Object to the Context, page 11-8, and Creating the Subscriber Object, page 11-9. The following C++ code fragment illustrates how this interface is obtained:

```
CORBA::Object_var dcObj =
    gateway_var->get("DeploymentContext",
        "admin", "admin");
```

This call will throw an exception if an invalid server name is given, or the user and passsword are not authorized. By analogy with the Naming Service, the returned object reference must be narrowed to the correct type, which in this case is `ATL_Deploy::DeploymentContext`:

```
ATL_Deploy::DeploymentContext_var dContext_var =
    ATL_Deploy::DeploymentContext::_narrow(dcObj);
```

Get ActionLauncher Object Reference from CorbaGatewayManager

The ActionLauncher interface is used to support the invocation of actions on the Cisco EMF system. The use of this interface is covered in Connecting the Subscriber to a Service Instance Using CORBA, page 11-11. A client will typically retrieve an object reference to an ActionLauncher object as part of its bootstrap process. The ActionLauncher object is obtained by using the get method of the CorbaGatewayManager interface, passing in `"ActionLauncher"` as the server name parameter. This method instantiates an ActionLauncher object and returns a reference to it. Here is a C++ code fragment to illustrate this:

```
CORBA::Object_var alObj =
    gateway_var->get("ActionLauncher",
        "admin", "admin");
```

This call will throw an exception if an invalid server name is given, or the user and password are not authorized. By analogy with the Naming Service, the returned object reference must be narrowed to the correct type, in this case `ATL_ACTION::ActionLauncher`:

```
ATL_ACTION::ActionLauncher_var act_var =
    ATL_ACTION::ActionLauncher::_narrow(alObj);
```

Get ObjectIDs for Class, Attribute and Containment Tree Names from AtlNaming

The CORBA Gateway uses opaque handles throughout its APIs to keep track of various objects which are not represented as CORBA objects. These handles are modeled in IDL using the ObjectID structure whose definition is as follows:

```
module ATL_OBJ
{
    struct ObjectID
    {
        unsigned long upper;
        unsigned long lower;
    };
};
```

`ATL_OBJ::ObjectID` is used to refer not only to managed objects, but also to attributes, object classes, and containment trees. For example, a Cisco EMF attribute is modeled in IDL as follows:

```
module ATL_OBJ
{
    struct Attr
        {
            ObjectID attrID;
            AttributeValueUnion attrValue;
        }
};
```

The attrID field identifies which attribute is concerned, and the attrValue will store an appropriately typed value (the AttributeValueUnion type is not expanded here for brevity, but consists of a union with different cases for each different type of attribute value).

While the opaque handle represented by the `ATL_OBJ::ObjectID` structure is used throughout the API set for efficiency reasons, its contents are by definition opaque, and consequently it is necessary for clients to retrieve the handle for an object based on its name. This is done using the AtlNaming interface, and it is typical for clients to retrieve the handles for all of the containment trees, object classes, and attribute names which their application requires once as part of their bootstrap code, and then use the handles as required by the interfaces. This may be done by using calls to the nameToId method of the AtlNaming interface, which is defined as follows:

```
short nameToId(
    in ATL_OBJ::ObjectNameList nl,
    in AtlType type,
    out ATL_OBJ::ObjectIDNamePairList successes,
    out ATL_OBJ::ObjectNameList failures)
```

where:

- `nl` is used to pass a list of the names for which ATL_OBJ::ObjectID handles should be returned.
- `type` is an enumerated parameter used to indicate the object type of the supplied names. A call to `nameToId` can pass names of only one type. For example, a list of allowed Managed object names would be flagged with CONTAINMENTOBJECTTYPE, attribute names would be flagged with

ATTRIBUTETYPE, object class names would be flagged with CLASSTYPE and containment tree names would be flagged with TREETYPE. The definition of the AtlType enumeration is shown below, and a table of descriptions of the values is given in Table 6-1 on page 6-2.

```
enum AtlType
    {
        CONTAINMENTOBJECTTYPE,
        ALARMTYPE,
        SECTIONTYPE,
        ATTRIBUTETYPE,
        CLASSTYPE,
        TYPETYPE,
        TREETYPE,
        TYPEACTIONTYPE,
    ACTIONSCENARIOTYPE
};
```

- `successes` is used to return the ATL_OBJ::ObjectID value for each name on the input names list which is successfully resolved. It is of type `ObjectIDNamePairList`, which is defined as follows:

```
module ATL_OBJ
{
    struct ObjectIDNamePair
    {
        ObjectID objectID;
        string objectName;
    }
    typedef sequence<ObjectIDNamePair>
    ObjectIDNamePairList;
    typedef sequence<string> ObjectNameList;
};
```

- `failures` returns any names from `nl` for which `ATL_OBJ::ObjectID` values could not be found.

For the purposes of this example, opaque handles can be retrieved at startup for the following names:

- The object classes Subscriber, Connection, and Service
- The attributes Name, Address, Ingress Port, Ingress VPI, Ingress VCI, Egress Port, Egress VPI, and Egress VCI
- The subscriberContainment containment tree

Managing Service Provision

This section describes the steps necessary to manage the provision of service to subscribers including:

- Creating the Subscriber Object Using CORBA, page 11-7
- Connecting the Subscriber to a Service Instance Using CORBA, page 11-11

Creating the Subscriber Object Using CORBA

This section describes the steps involved in creating the Cisco EMF representation of a subscriber. This involves creating a subscriber object with the appropriate attribute values using the deployment CORBA interfaces.

Adding the Subscriber Object to the Context

This section describes how a subscriber object is added to the deployment context. This is done using the addObjectDefinition method of the DeploymentContext interface, which is defined as follows:

```
ATL_OBJ::ObjectID addObjectDefinition(
    in octet objectType,
    in ATL_OBJ::ObjectID classID,
    in ContainmentSpecVector containments,
    in ATL_OBJ::AttributeList attributes,
    in ATL_OBJ::AttributeList participantAttributes,
    in string name)
```

where the parameters to this call are:

- `objectType` refers to the type of object to be created. The list of object types and their corresponding `objectType` values are defined in the Attribute.idl file. For example, to create managed objects, this should be `ATL_OBJ::MANAGED_OBJECT`.
- `classID` specifies the object class to be created. This should be set to the opaque handle for the Subscriber class retrieved as described in Get ObjectIDs for Class, Attribute and Containment Tree Names from AtlNaming, page 11-6. The class name for the Subscriber object class is Subscriber.
- `containments` specifies containment relationships for the subscriber object. ContainmentSpecVector is defined in the IDL `ATL_Deploy` module as shown here:

```
struct ContainmentSpec
{
    enum ContainmentStatus
    {
        Path_unprocessed,
        Path_okay,
        Path_invalidParent,
        Path_badTree,
        Path_error
    } status;
    ATL_OBJ::ObjectID treeID;
    ATL_OBJ::ObjectID parentID;
    string name;
};
typedef sequence<ContainmentSpec>
ContainmentSpecVector;
```

To create the subscriber requires only one ContainmentSpec, with values as follows:

- `status` should be set to Path_unprocessed
- `treeID` should be set to the opaque handle for the subscriber containment tree retrieved as described in Get ObjectIDs for Class, Attribute and Containment Tree Names from AtlNaming, page 11-6. The containment tree name for the subscriber containment tree is subscriberContainment.
- *parentID* should be set to the opaque handle for the port managed object retrieved as described in Get ObjectIDs for Class, Attribute and Containment Tree Names from AtlNaming, page 11-6. The managed object name of the port object will be in the form `"subscriberContainment:/<network>/<port>"` with the names of the relevant objects in the hierarchy substituted for values enclosed in <>s.
- `name` for the subscriber object.

- `attributes` specifies initial values for attributes of the object. The Attribute structure is defined as follows:

```
module ATL_OBJ
{
    struct Attr
    {
        ObjectID attrID;
        AttributeValueUnion attrValue;
}
    typedef sequence<Attribute> AttributeList;
};
```

  Values may be supplied for the following attributes:

  – Name

  – Address

  – The participantAttributes parameter need not be used for the creation of the subscriber object

  – The name parameter specifies the name of the subscriber object

  The return value of the method is an opaque handle for the object definition added to the context. It must be noted that this is not the same as the managed object opaque handle which will be generated when the object is created. This handle is valid only for the lifetime of the context.

Creating the Subscriber Object

The subscriber object added to the context is created using the startContext method of the ParticipationCoordinator interface.

```
void startContext( in CorbaParticipationContext cpc,
    in short progressReportFlag,
    in long startLevel,
    in long stopLevel,
    in Initiator initiator,
    in unsigned long userData)
```

where the parameters to this call are:

- `cpc` is the DeploymentContext object (the DeploymentContext interface inherits from CorbaParticipationContext).
- `progressReportFlag` allows the level of progress reporting to be set. For creating the subscriber objects this may be set to 0, that is only completion of the operation will be reported.
- `startLevel` should be set to the IDL defined constant value Provision_Level_Base for all object creation.
- `stopLevel` should be set to the IDL defined constant value Provision_Level_End for all object creation.
- `initiator` is an object reference for the Initiator object on the client side which should be called back when the creation of the objects described in the context is complete. The contextComplete method of the Initiator object will be invoked when the deployment cycle is complete for the context. This interface is described following the description of the parameters for the startContext call.
- `userData` will be passed as a parameter to the contextComplete call on the initiator object. It is intended to provide easier client side correlation between startContext calls and the resulting contextComplete callbacks, when there are multiple outstanding requests.

When the context is complete, the coordinator will call back the initiator with a contextComplete call.

```
void contextComplete( in CorbaParticipationContext cpc,
        in ParticipationInfo pInfo,
        in InitiatorCallback iCallback,
        in unsigned long userData)
```

where the parameters to this call are:

- `cpc` provides an object reference for the context which has completed. The status attribute of the context may be checked to see whether the objects in the context were created successfully. CorbaParticipationContext is defined in IDL as follows:

```
module ATL_PART
{
    enum ParticipationStatus
    {
        Success,
        PartialFailure,
        Failure,
        Status_unknown
};
        enum ParticipationDirection
    {
            Context_going_forward,
            Context_going_backward
    };
        interface CorbaParticipationContext
{
readonly Attr long contextID;
    attribute ParticipationStatus status;
    attribute ParticipationDirection direction;
};
};
```

- `pInfo` provides some background information on the processing of the context. ParticipationInfo is defined:

```
module ATL_PART
{
    struct ParticipationInfo
    {
        long numberParticipants;
        long levelsVisited;
    };
};
```

- `iCallback` is an object reference for the InitiatorCallback which should be notified when the client is finished with the context object. See Destroying the Context, page 11-10 for more details.
- `userData` is the same value which was initially passed to the corresponding startContext method call.

Destroying the Context

The final step required of the client is to notify the Cisco EMF server that it is finished with the context to allow the context object to be discarded. This is done using the contextCompleteCB method call of the InitiatorCallback object passed as the iCallback parameter to the contextComplete call.

```
void contextCompleteCB( in long contextID,
    in ParticipationStatus status,
    in unsigned long userData)
```

where:

- `contextID` is the identifier for the context which is no longer required. The identifier can be retrieved from the contextID attribute of the context (defined as part of the CorbaParticipationContext interface, see Creating the Subscriber Object, page 11-9).
- `status` can be derived from the status attribute of the context (defined as part of the CorbaParticipationContext interface, see Creating the Subscriber Object, page 11-9).
- The `userData` parameter should be set to the same value that was received as the userData parameter in the contextComplete call.

Connecting the Subscriber to a Service Instance Using CORBA

A subscriber is connected to a service using the invokeAction method of the ActionLauncher interface defined below:

```
module ATL_ACTION
{
    struct Action
    {
        string actionName;
        unsigned long time;
    };
    struct ObjectTypeDescriptor
    {
        ATL_OBJ::ObjectIDList objectIDs;
        ATL_OBJ::AttributeList attributes;
    };
    typedef sequence<ObjectTypeDescriptor>
        ObjectTypeDescriptorList;
    struct ActionScenario
    {
        Action action;
        ObjectTypeDescriptorList otds;
    };
        interface ActionResultCallback;
        interface ActionLauncher
    {
short invokeAction( in ActionScenario as,
        in ActionResultCallback cb,
        in unsigned long userData);
    };
};
```

The parameters of the invokeAction method are as follows:

- `as` specifies the details of the action to be carried out. In the context of connecting a subscriber, the following values should be set within the ActionScenario:
  - action.actionName = “connect”
  - action.time = the time when the action should be carried out.
- `otds` should include two ObjectTypeDescriptors:
  - `ObjectTypeDescriptor 1`

    `objectIDs` contains the opaque handle for the subscriber object. This may be obtained from the context after the subscriber has been created using the method `getObjectIDForObjectDefinition` of the DeploymentContext interface. The value for the

`objDefKey` parameter used should be the value returned when the object definition was added to the context, as described in Adding the Subscriber Object to the Context, page 11-8. The call returns the opaque handle for the subscriber object.

Alternatively, the opaque handle for the subscriber object may be obtained as described in Get ObjectIDs for Class, Attribute and Containment Tree Names from AtlNaming, page 11-6. The name of the object will be in the form `"subscriberContainment:/<Network>/<port>/<subscriber>"` with the names of the relevant objects in the hierarchy substituted for values enclosed in <>s.

`attributes` should contain values for Ingress VPI and Ingress VCI, representing the information for the incoming ATM connection from the subscriber.

– `ObjectTypeDescriptor 2`

`objectIDs` contains the opaque handle for the Service object.

The opaque handle for the Service object may be obtained as described in Get ObjectIDs for Class, Attribute and Containment Tree Names from AtlNaming, page 11-6. The name of the object will be in the form `"subscriberContainment:/<network>/<port>/<service>"` with the names of the relevant objects in the hierarchy substituted for values enclosed in <>s.

`attributes` should be empty.

`cb` is an object reference for the callback object on the client side which should have its actionResult method invoked when the action has completed. The ActionResultCallback interface is described following this bullet list of parameter descriptions for the invokeAction method.

The userData parameter is intended to simplify client side correlation of actionResult calls with invokeAction calls. The value of this parameter will be passed to the actionResult method when it is invoked on the callback object.

The actionResult method of the ActionLauncherCallback interface, which is called to communicate the result of an invokeAction call, is defined below:

```
module ATL_ACTION
{
    typedef sequence<string> Annotations;
    struct ActionResult
    {
        boolean status;
        ATL_OBJ::ObjectAttributeList oal;
        Annotations annotations;
    };
    interface ActionResultCallback;
    {
        void actionInvocationResult( in ActionResult result,
        in unsigned long userData);
    };
};
```

The parameters of the actionResult method are as follows:

- The `result` parameter provides the output from the action.
- The `status` field of the result is a boolean indicating whether the action succeeded.
- The `oal` field will contain a single `ObjectAttributeListEntry`, defined below, whose `id` field will be the id of the created connection, and whose `attrs` and `fails` fields will be empty.

- The `annotations` field contains a set of textual messages representing the status of the action.

```
module ATL_OBJ
{
    struct ObjectAttributeListEntry
    {
        ObjectID id;
        AttributeList attrs;
        AttributeList fails;
    };
    typedef sequence<ObjectAttributeListEntry>
        ObjectAttributeList;
};
```

- The `userData` parameter is the value which was passed as userData to the invokeAction method call described earlier in this section.

Managing Service Removal

This section describes how a subscriber's connection to a service may be torn down. This is in fact very similar to the process of providing service to the subscriber. The process for providing service was described in Managing Service Provision, page 11-7.

The process can be divided into two steps:

- Disconnecting the Subscriber from a Service Instance Using CORBA, page 11-13
- Destroying the Subscriber Object Using CORBA, page 11-14

Disconnecting the Subscriber from a Service Instance Using CORBA

This is similar to the act of connecting the subscriber to the service instance, described in Connecting the Subscriber to a Service Instance Using CORBA, page 11-11. Both acts make use of the invokeAction method of the ActionLauncher interface. The difference between them is in the contents of the as ActionScenario parameter, which for disconnection should be as follows:

- action.actionName = "disconnect"
- action.time = the time at which the action should be carried out.
- otds should include one ObjectTypeDescriptor as follows:

  objectIDs contains the opaque handle for the connection object. This may be obtained as described in Get ObjectIDs for Class, Attribute and Containment Tree Names from AtlNaming, page 11-6. The name of the object will be in the form `"subscriberContainment:/<network>/<port>/<subscriber>/<connection>"` with the names of the relevant objects in the hierarchy substituted for values enclosed in <>s. The <connection> name is automatically generated by the connect action described in Connecting the Subscriber to a Service Instance Using CORBA, page 11-11, and will be in the form `"<subscriber>-<service>"` with the names of the relevant objects substituted for the values enclosed in <>s. attributes should be empty.

Destroying the Subscriber Object Using CORBA

This is similar to the creation of the Subscriber object described in Creating the Subscriber Object Using CORBA, page 11-7. The only differences are as follows:

- The addObjectDefinitionForDeletion method should be used to add the object to the context.
- No attributes need to be set when adding the Subscriber object to the context.
- When invoking the startContext method (described in Creating the Subscriber Object, page 11-9), different values should be used for the startLevel and stopLevel parameters as follows:
  - The startLevel parameter should be set to the IDL defined constant value `Deletion_Level_Base` for all object destruction.
  - The stopLevel parameter should be set to the IDL defined constant value `Deletion_Level_End` for all object destruction.

Service Modification via CORBA

This section describes how subscribers' connections to services may be modified via CORBA. As the service instance objects are to be created manually, changes to the service instance object via CORBA are not described. Consequently, the changes described here are restricted to the subscriber object.

This section includes the following information:

Get DataAbstractor Object Reference from CorbaGatewayManager

The DataAbstractor interface allows clients to access and modify the values of attributes of objects in the Cisco EMF system. The DataAbstractor object is obtained by using the get method of the CorbaGatewayManager interface, passing in `"DataAbstractor"` as the serverName parameter. Here is a C++ code fragment to illustrate this:

```
CORBA::Object_var anObj =
    gateway_var->get("DataAbstractor",
        "admin", "admin");
```

This call will throw an exception if an invalid server name is given, or if the user and password are not authorized. By analogy with the Naming Service, the returned object reference must be narrowed to the correct type, in this case `ATL_DABS::DataAbstractor`:

```
ATL_DABS::DataAbstractor_var dabs_var =
    ATL_DABS::DataAbstractor::_narrow(anObj);
```

Modification of the Subscriber Object

The subscriber object has the following attributes which may be modified:

- Name
- Address

These may be modified for a given subscriber or subscribers using the setAsync method of the DataAbstractor interface, defined in IDL as:

```
void setAsync(
       in ATL_OBJ::ObjectAttributeList requests,
       in DataAbstractorCallback daCallback,
       in unsigned long userData)
```

This method makes use of the following supporting definitions:

- The requests parameter specifies the changes to be made. The requests parameter is made up of a sequence of ObjectAttributeListEntry structures. One of these structures should be used for each object to be changed. The id field of the structure should be set to the opaque handle for the object to be modified. The attrs field of the structure should contain the desired attribute values for the object. The fails field of the structure should be empty: it is used by the DataAbstractor when returning information to the caller. The structure `ObjectAttributeListEntry` and sequence `ObjectAttributeList` are defined in IDL as follows:

  ```
  module ATL_OBJ
  {
  struct ObjectAttributeListEntry
      {
          ObjectID id;
          AttributeList attrs;
          AttributeList fails;
      };
      typedef sequence<ObjectAttributeListEntry>
  ObjectAttributeList;
  };
  ```

- The daCallback parameter is an object reference for the callback object on the client side which should have its setCB method invoked when the set has completed. The setCB method of the DataAbstractorCallback interface is defined below this bullet list.
- The userData parameter is intended to simplify client side correlation of setCB calls with setAsync calls. The value of this parameter will be passed to the setCB method when it is invoked on the daCallback object.

Here is the IDL definition of the setCB method of the DataAbstractorCallback interface:

```
module ATL_DABS
{
    interface DataAbstractorCallback
    {
        ...
        void setCB(
    in ATL_OBJ::ObjectAttributeList results,
    in unsigned long userData);
        ...
    };
};
```

The setCB method is invoked to notify the client that a setAsync call to the DataAbstractor interface has completed. The parameter values are as follows:

- The results parameter will contain basically the same information as was passed in the requests parameter to the setAsync call. The only difference is that any attributes for which the set failed will appear on the fails attribute list of the relevant object attribute list entry rather than the attrs attribute list.
- The userData parameter will be the same value passed as the userData parameter to the setAsync call. This assists the correlation of requests and results of asynchronous calls.

Troubleshooting and Helpful Hints

Troubleshooting

Table 12-1 Problem: Client fails to connect to CORBA Gateway servers

Possible cause	Remedy
Orbix daemon not running	Check that the Orbix daemon is running on the Cisco EMF server machine, and restart if required.
Orbix daemon port number mismatch	Check the port number configuration of the client machine. For Orbix clients, the IIOP port should be set the same as the listening port on the server machine (1570). Beware OrbixWeb 3.0, which as default sets this to 1571!

Table 12-2 Problem: Client fails to connect to Naming Service

Possible cause	Remedy
Naming Service configuration problems	Check NS_HOST is set to the correct machine name in the client configuration.

Table 12-3 Problem: Orbix Daemon crashes with Naming Service errors

Possible cause	Remedy

Multiple zombie copies of the Naming Service may be running simultaneously.	Check using UNIX command
	`ps -ef | grep jre`
	and not the Orbix command
	`psit!`
	Kill all copies of the Naming Service, then restart.

Table 12-4 Problem: Client receives no callbacks

Possible cause	Remedy
Client code is not processing the CORBA event queue.	Ensure that the client code has enabled event processing, such as entering an event loop.
Client code linked only with client stubs.	Clients which implement callback objects need to be linked with both client-side and server-side ORB code.

Hints

1. Unsure of the signatures of the methods which must be implemented for client-side callback class? Run the Orbix IDL compiler with the **-S** option.
2. To check which CORBA servers are running on the CORBA Gateway: use the Orbix utility command: **psit**. If this fails, check that the Orbix daemon **orbixd** is running.
3. To check that the Naming Service is running, and has registered the name of the CorbaGatewayManager correctly: use the Orbix utility command **lsns**.

Other Resources

Recommended Reading

Advanced CORBA Programming with C++

Michi Henning, Steve Vinoski
Addison-Wesley, 1999
ISBN 0201379279

Instant CORBA

Robert Orfali, Dan Harkey, Jeri Edwards
Wiley, 1997
ISBN 0471183334

CORBA Distributed Objects, using Orbix

Sean Baker
ACM Press, Addison-Wesley, 1997
ISBN 0201924757

CORBA for Dummies

John Schettino, Liz O'Hara
IGD Books, 1998
ISBN 0764503081

The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture

Allan Pope
Addison-Wesley, 1998
ISBN 0201633868

Client/Server Programming with Java and CORBA, 2nd Edn

Robert Orfali, Dan Harkey
Wiley, 1998
ISBN 047124578X

The Essential Distributed Objects Survival Guide

Orfali, Harkey, Edwards
Wiley, 1996
ISBN 0471129933

CORBA Design Patterns

Raphael C. Malveau and Thomas J. Mowbray,
Wiley, 1997
ISBN 0471158828

Notification Service

OMG TC Document telecom/99-07-01
Joint Revised Submission
OMG

OrbixNotification Programmer's Guide and Reference

IONA Technologies PLC
November 1998

WWW Resources

http://www.omg.org—The OMG website. Contains a library of the CORBA specifications, and news of developments.

http://www.iona.com—Iona Technologies. Information on the Orbix ORB, including a developers' area and knowledge base.

http://www.cetus-links.org—Extensive collection of Object-Orientated internet resources, of particular interest are the list of ORBs, and links to information on the use of CORBA in telecommunications.

A

Asynchronous Calls

When an asynchronous method call is made, the call returns immediately. The result of the method call is returned to the calling process at some later time. The advantage of this is that the caller is not blocked while waiting for the reply (which could take some time). The disadvantage is that the coding for the asynchronous callback mechanism is not as simple as for a synchronous call. There are various ways in which to implement asynchronous messaging in CORBA - the method chosen for Cisco EMF is that of callbacks. The caller has to provide a callback object with appropriate methods which the server can invoke to reply, hence the caller becomes a server which has to handle incoming requests with an appropriate event loop mechanism.

See also Oneway Calls

C

Cisco EMF

The Cisco Element Management Framework Carrier Class Framework is a product suite of advanced carrier class framework, end operator applications, and toolkits. Cisco EMF is used to quickly develop and deploy element, network and service-level applications in broadband access technologies ranging from Digital Subscriber Line (DSL), used for high speed internet access, cable modems, Voice Over IP, to complex ATM/IP routing multi-service switches.

Client

A client is any entity which is able to request a service. A CORBA client could itself be a CORBA server, but while invoking a method on another CORBA object, it is said to be acting as a client.

See also Server

CORBA (Common Object Request Broker Architecture)

CORBA has been developed by the OMG to address the need for industry standards for middleware. The CORBA specification includes an interface definition language (IDL), which is a language-independent way of creating contracts between objects for implementation as distributed applications

D

Deployment

The Cisco EMF interface for creation, configuration and deletion of Network Objects, using the Generic Participation Framework. Sometimes also referred to as ‘Provisioning’, ‘Pre-provisioning’ or ‘Data-building’.

E

Element Manager — A Cisco EMF Element Manager is an application which is responsible for providing FCAPS (Fault, Configuration, Accounting, Performance, and Security) management for a particular type of network element or family of network elements. In this respect element managers are technology-specific whereas the Cisco EMF is technology-independent.

Exception (in IDL) — An IDL construct that represents an exceptional condition that could be returned when an invocation results in an error. There are two categories of exceptions; system exceptions and user-defined exceptions.

F

Factory Object — A CORBA object that is used to create new CORBA objects, as and when required by client activity. Factory objects are themselves usually created at server installation time.

G

Generic Participation Framework — A mechanism designed to permit generic clients (CORBA or otherwise) to participate in Cisco EMF object lifecycle operations.

I

IDL (Interface Definition Language) — An abstract programming language used to define CORBA interfaces. The interface is the description of the complete set of operations that a client can invoke on that server. IDL is independent of the actual programming language used to implement either server or client - it must be processed by an IDL compiler to generate code.

IDL Compiler — A tool supplied by an ORB vendor or other supplier that takes an interface written in IDL and produces programming language interfaces, structures or classes as appropriate, which are then used to develop the clients and servers in the chosen language.

See also Stubs and Skeletons

.IIOP (Internet InterORB Protocol) — The OMG-specified network protocol for communicating between ORBs.

Implementation Class — A concrete class that implements the behavior for all of the operations and attributes of the IDL interface it supports.

IOR (Interoperable Object Reference) — An interoperable object reference is generated by the ORB whenever an object reference is passed between ORBs. It contains self-describing data identifying the ORB domain of the object and the communication protocols it supports. The application developer does not normally need to worry about the internal contents of an IOR. See also Stringified IOR

N

Naming Service

A CORBA service that allows CORBA objects to be named by means of binding a name to an object reference. A client can then supply this name to obtain the desired object reference.

O

Object

A computational grouping of operations and data into a modular unit. Since there are different conceptual types of objects which are involved in the Cisco EMF-CORBA Gateway, described below:

CORBA object—An object which is defined by an IDL interface, and for which an object reference is available. Each CORBA object is basically broken into two pieces; a client-side proxy and a server-side implementation. The server side implementation contains the application logic. The client-side proxy is a proxy for the server-side object, and forwards method calls to the server-side implementation.

Cisco EMF object—Not just managed network objects are objects in Cisco EMF, every attribute, class, event, containment tree etc. is an object, identified by a unique, generic ObjectID.

Managed Network object—This represents a network element, module or device.

ObjectID (Cisco EMF Generic Object Identifier)

Every managed object, attribute, class, etc. in Cisco EMF is assigned a unique Object Identifier. This is an opaque handle which is represented externally by the IDL structure ATL_OBJ::ObjectID.

Object Reference

A value which reliably identifies a particular object. A CORBA Object Reference is a construct containing the information needed to specify an object within an ORB domain. Object references are the CORBA object equivalent to programming language-specific object pointers. There are both vendor-specific, and vendor-independent interoperable object references.

See also IOR (Interoperable Object Reference) and Stringified IOR

OMG (The Object Management Group)

A consortium of over 700 companies and organizations which works to establishes industry guidelines and object management specifications for distributed object-orientated computing, including the CORBA standard. See the OMG website (http://www.omg.org) for further information.

ORB (Object Request Broker)

The ORB infrastructure enables CORBA objects to communicate with each other. Applications developed using ORBs conforming to the CORBA 2.0 standard or higher are able to interoperate with each other. Currently, more than 40 ORBs are available from various commercial vendors and other organizations.

Oneway Calls

The ‘oneway’ IDL keyword preceding a method definition is used to denote that the method returns without blocking. Except in exceptional circumstances.

This is important in situations where single-threaded clients and servers may be messaging each other simultaneously to avoid deadlocks. It is used in some of the asynchronous interfaces to avoid clients waiting for lengthy server-side processing to complete. It has the disadvantage that the client does not know if the call succeeded, as no status return or exception is allowed.

P

Provisioning See Deployment

S

Server A server provides a service which is available to other entities. A CORBA server is an entity which implements one or more IDL interfaces. Servers are sometimes referred to as object servers.

See also Client.

Stringified IOR An IOR which has been converted to a string so that it may be stored in a text file. Such strings should be treated as opaque. Standard object_to_string() and string_to_object() methods are part of the CORBA specification.

See also IOR (Interoperable Object Reference)

Stubs and Skeletons The code which is generated from an IDL interface by an IDL compiler. Stubs provide the remote CORBA object functionality on the client side - from the client application's point of view, they appear to be local calls. Skeletons perform a similar function for the server. They hide the details of the data marshalling and transmission from the application.

U

UML (Unified Modeling Language) A modeling language that incorporates key concepts of earlier modeling methodologies: Booch, OMT, and OOSE.

A

C

D

E

F

G

I

M

N

O

P

R

S

T

U

W