IONA

Orbix®

Enterprise Messaging Guide
Java

Version 6.1, December 2003

Making Software Work Together™

Contents

Part 1   Messaging Service Technologies

Part 2   The Notification Service

Part 3 The Telecom Log Service

Part 4 The Java Messaging Service

Part 6 The JMS-Notification Bridge Service

List of Tables

LIST OF TABLES

List of Figures

LIST OF FIGURES

Preface

Audience

This guide is intended to help you become familiar with the notification
service, and shows how to develop applications with it. This guide assumes
that you are familiar with CORBA concepts, and with Java.

This guide does not discuss every interface and its operations in detail, but
gives a general overview of the capabilities of the notification service and
how various components fit together. For detailed information about
individual operations, refer to the *CORBA Programmer's Reference*.

Related Documentation

For the latest version of all IONA product documentation, see the IONA web
site:

http://www.iona.com/docs/

Organization of this Guide

Read "Messaging Service Technologies" for an overview of the Orbix enterprise messaging services. Subsequent parts describe various components of the messaging service in detail, and show how you implement an application that uses its capabilities.

Document Conventions

This guide uses the following typographical conventions:

Constant width	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.
	Constant width paragraphs represent code examples or information a system displays on the screen. For example:
	`#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*.
	Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:
	`% cd /users/`*your_name*
	Note: some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, Windows95, or Windows98 command prompt.

. . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[]	Brackets enclose optional items in format and syntax descriptions.
{}	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in {} (braces) in format and syntax descriptions.

Part 1

Messaging Service Technologies

Overview

Orbix provides enterprise messaging technology through the CORBA notification service, the CORBA telecom log service, and the Java Messaging System(JMS). Orbix also provides a bridging service that allows the CORBA notification service and JMS to seamlessly share messages.

In this part

This part contains the following chapters:

CORBA Messaging Technologies

Overview

The arcitecture of the CORBA event service provides the foundation for the CORBA messaging technologies. In the event service, client *suppliers* generate messages which are forwarded to client *consumers* through an *event channel*. The event channel provides a mechanism for publish / subscribe messaging, but does not support point-to-point messaging.

The notification service provides enterprise level decoupled messaging facilities by extending the functionality of the CORBA event service to include Qualities of Service, subscription mechanisms, filtering, and structured messages.

The telecom log service encompasses the functionality of both the event service and the notification service and extends their functionality by adding a durable and searchable log. The logs record the events forwarded through the associated event or notification service.

> **Note:** The telecom log service also provides a log for non-messaging CORBA clients.

In this chapter

This chapter discusses the following topics:

Event Service

Overview

An event originates at a client *supplier* and is forwarded through an *event channel* to any number of client *consumers*. Suppliers and consumers are completely decoupled; a supplier has no knowledge of the number of consumers or their identities, and consumers have no knowledge of which supplier generated a given event.

Service Capabilities

An event channel provides the following capabilities for forwarding events:

- Accepts incoming events from client suppliers.
- Forwards supplier-generated events to all connected consumers.

Connections

Suppliers and consumers connect to an event channel and not directly to each other, as shown in Figure 1. From a supplier's perspective, the event channel appears as a single consumer; from a consumer's perspective, the event channel appears as a single supplier. In this way, the event channel decouples suppliers and consumers.
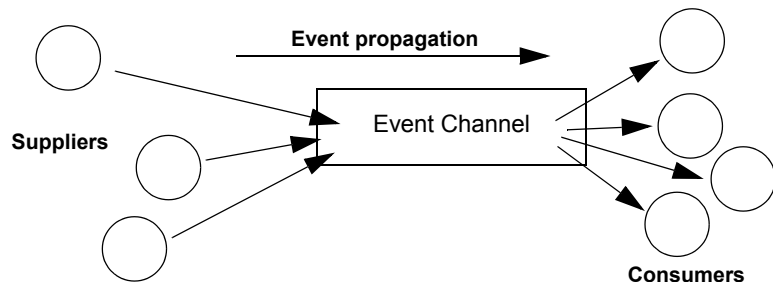
Figure 1: *Suppliers and Consumers Communicating through an Event Channel*

How Many Clients?

Any number of suppliers can issue events to any number of consumers using a single event channel. There is no correlation between the number of suppliers and the number of consumers. New suppliers and consumers can be easily added to or removed from the system.

Example

Many documents can be linked to a spreadsheet cell, and must be notified when the cell value changes. However, the spreadsheet software does not need to know about the documents linked to its cell. When the cell value changes, the spreadsheet software should be able to issue an event that is automatically forwarded to each connected document.

Event Delivery

Figure 2 shows a sample implementation of event propagation in a CORBA system. In this example, suppliers are implemented as CORBA clients; the event channel and consumers are implemented as CORBA servers. An event occurs when a supplier invokes a clearly defined IDL operation on an object in the event channel application. The event channel then propagates the event by invoking a similar operation on objects in each of the consumer servers.

Supplier

Consumers

Event Channel

1. Supplier calls operation on event channel

2. Event channel calls operation on consumers
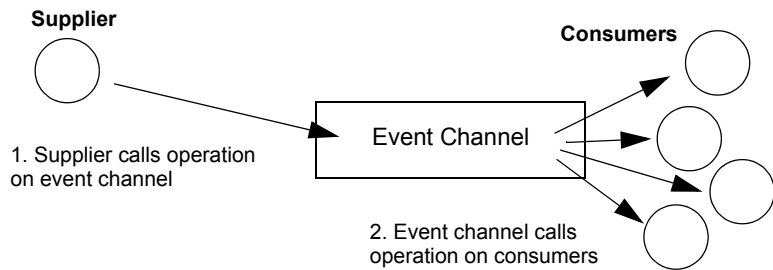
Figure 2: *Event Propagation in a CORBA System*

Further Reading

For a full discussion of the event service and how to develop applications with it see the *CORBA Programmer's Guide*.

Notification Service

Extensions of Event-based Communication

The notification service extends the concept of event-based messaging with the following features:

Feature	Description
Quality-of-service	Properties such as event message priority and lifetime, can be set on different levels within the event channel.
Persistence	Quality-of-service parameters control the availability of events and channels beyond the lifetime of the service process, supplier processes, or consumer processes.
Event filtering and subscription	Filters allow consumers to receive only the events they are interested in, and to tell suppliers which events are in demand.
Event publication	Suppliers can inform an event channel which events they can supply, so consumers can subscribe to new event types as they become available.
Structured events	Header information in structured events let you set properties and filterable data on event messages.
Multicast event delivery	Groups of consumers can subscribe to events and receive them using UDP multicast protocol, which keeps network traffic to a minimum.

For more information on the CORBA notification service, see "The Notification Service" on page 19

Telecom Log Service

Overview

The telecom log service is modeled on the CORBA notification service and uses event-aware objects and an event channel to manage the logging of events to a persistent store. This implementation allows logs to generate events relating to the log and propagate them to their clients, filter events for logging, and forward events from suppliers to consumers. It also allows notification channel-aware logs to leverage the notification service's Quality of Service (QoS) properties. The telecom log service also provides interfaces that allow event-unaware clients to write directly to the log.

Figure 3 shows a basic telecom log service configuration.
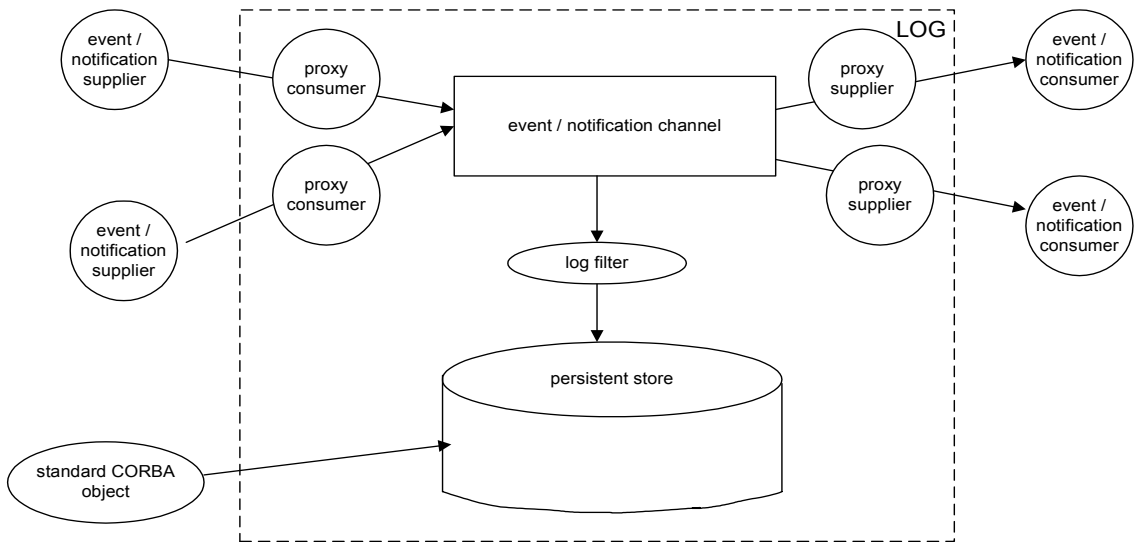
Figure 3: *Log service configuration*

Features of the Telecom Log Service

The telecom log service offers the following extensions to the notification service:

Table 1:   *Features of the telecom log service*

Feature	Description
Log generated events	Log objects can keep their event aware clients informed of the telecom log service's state by generating events and forwarding the events onto their clients.
Quality of Service	The telecom log service specifies three levels of Quality of Service for logged events.
Log size	The size of the persistent store for each log object can be set individually.
Log full behavior	The behavior of the log when it becomes full is configurable. The log can either discard new log records until the old ones are deleted manually, or the log can overwrite the oldest records in the store with new ones.
History	The maximum lifetime of a log record can be controlled through property settings.
Scheduling	Record logging can be scheduled. When the log object is scheduled to log events, it is fully functional. When it is not scheduled to receive events, the log object will continue to provide read access to the logged events and perform the functions of an event or notification channel.
Filtering	In addition to delivery level filtering, NotifyLog objects support event filtering at the logging level. They can apply filters to the events that are recorded in the log's persistent store.

For more information of the telecom log service, see .

Event Communication

Overview

CORBA specifies two approaches to initiating the transfer of events between suppliers and consumers

- *push model*: Suppliers initiate transfer of events by sending those events to the channel. The channel then forwards them to any consumers connected to it.
- *pull model*: Consumers initiate the transfer of events by requesting them from the channel. The channel requests events from the suppliers connected to it.

Push Model

In the push model, suppliers generate events and actively pass them to an event channel. In this model, consumers wait for events to arrive from the channel.

Figure 4 illustrates a push model architecture in which push suppliers communicate with push consumers through the event channel.
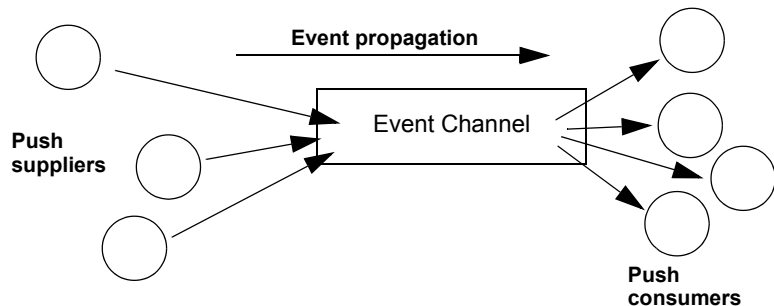
Figure 4: *The Push Model of Event Transfer*

Pull Model

In the pull model, a consumer actively requests events from the channel. The supplier waits for a pull request to arrive from the channel. When a pull request arrives, event data is generated and returned to the channel.

Figure 5 illustrates a pull model architecture in which pull consumers communicate with pull suppliers through the event channel.
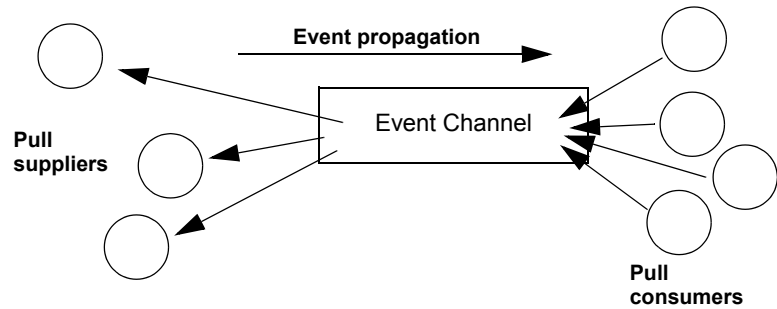
Figure 5: *Pull Model Suppliers and Consumers Communicating through an Event Channel*

Mixing Push and Pull Models

Because suppliers and consumers are completely decoupled by the event channel, push and pull models can be mixed in a single system.

For example, suppliers can connect to an event channel using the push model, while consumers connect using the pull model, as shown in Figure 6.
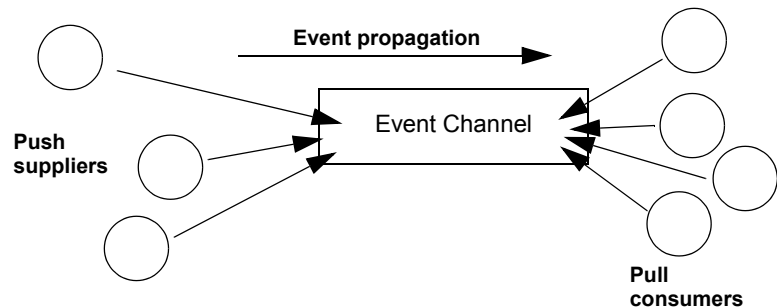
Figure 6: *Push Suppliers and Pull Consumers Communicating through an Event Channel*

In this case, both suppliers and consumers participate in initiating event transfer. A supplier invokes an operation on an object in the event channel to transfer an event to the channel. A consumer then invokes another operation on an event channel object to transfer the event data from the channel.

In the case where push consumers and pull suppliers are mixed, the event channel actively propagates events by invoking IDL operations in objects in both suppliers and consumers. The pull supplier would wait for the channel to invoke an event transfer before sending events. Similarly, the push consumer would wait for the event channel to invoke event transfer before receiving events.

The Java Messaging Service

The Java Messaging Service (JMS) provides a native mechanism for Java applications to participate in messaging systems.

Overview

Orbix provides messaging services to its J2EE application server through an implementation of Sun's Java Messaging Service(JMS) specification. Iona's JMS implementation can also provide messaging services to any application written in Java, including CORBA applications.

In this chapter

This chapter discusses the following topics:

Java Messaging Service Overview

Messaging architecture

In general, JMS clients can be either a *MessageProducer* or a *MessageConsumer*. Producers send messages to a *Destination* and consumers receive the messages from the destination. Figure 7 shows a simple JMS system with one producer and one consumer.

Figure 7: *Simple JMS system*

Java Messaging Service Features

The Java Messaging Service provides the following features for messaging:

- Point to point messaging
- Publish / subscribe messaging
- Quality of Service properties to set the priority and lifetime of messages
- Properties to set the level of guarantee that a message will be delivered
- Information to associate messages with one another
- User definable properties
- Header information defining a reply destination
- Filtering

Point to Point Messaging

Architecture

When using the point to point messaging, JMS producers, *QueueSenders*, and JMS consumers, *QueueRecievers*, connect to a specialized destination called a *Queue*. Producers place messages on the queue and the messages are consumed in FIFO order by message consumers. Once a message has been consumed it is removed from queue.

Figure 8 shows a JMS point to point implementation.
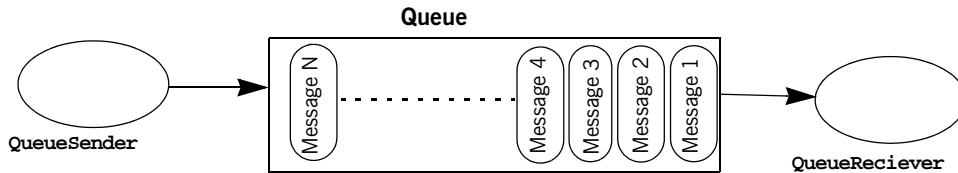
Figure 8: *JMS Point to Point Messaging*

JMS queues ensure that messages are delivered to one message consumer. If no consumer is connected to the queue messages are stored until one connects.

Publish / Subscribe Messaging

Architecture

JMS also supports publish / subscribe style messaging which functions similarly to the CORBA notification service. Many producers, `TopicPublishers`, and consumers, `TopicSubscribers`, connect to a destination, called a `Topic`. The topic receives messages from all connected producers and forwards the messages to all consumers. Consumers can opt to not receive certain messages by implementing a `MessageSelector` to filter out messages.

Figure 9 shows a JMS publish / subscribe implementation.
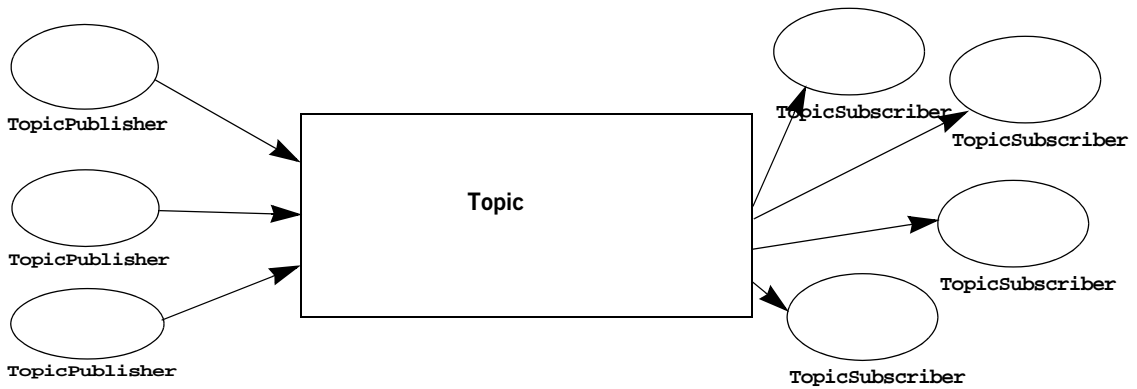
Figure 9: *JMS Publish/Subscribe Messaging*

Delivery guarantee

JMS topics offer no guarantee that messages will be delivered to any consumers. If there are no consumers connected to the topic when a message is sent, the message is simply dropped. If consumers wish to ensure that all messages are delivered to them regardless of their connections status, the consumer can register a *durable* subscription.

For more information on the Java Messaging Service, see "The Java Messaging Service" on page 231.

The JMS-Notification Bridge Service

The bridge service allows JMS and CORBA notification clients to share messages.

Overview

Orbix provides a bridging mechanism between CORBA notification service clients and JMS clients. Using this bridge notification service clients and JMS clients can exchange messages based on the OMG's *Notification / Java Message Service* specification.

Message sharing

The JMS-Notification bridge allows JMS publishers to forward messages to CORBA notification consumers and CORBA notification suppliers to forward messages to JMS subscribers. This is done using unidirectional bridges that mimic JMS and notification clients.

For example, a bridge forwarding messages from a notification channel to a JMS topic acts as a CORBA notification consumer and a JMS publisher. Figure 10 shows an application that uses two bridges to facilitate bidirectional messaging between a JMS topic and a notification channel. All

of the events supplied to the notification channel are forwarded to the notification consumers and `Bridge2`. All of the messages published to the JMS topic are forwarded to the subscribers and `Bridge1`.
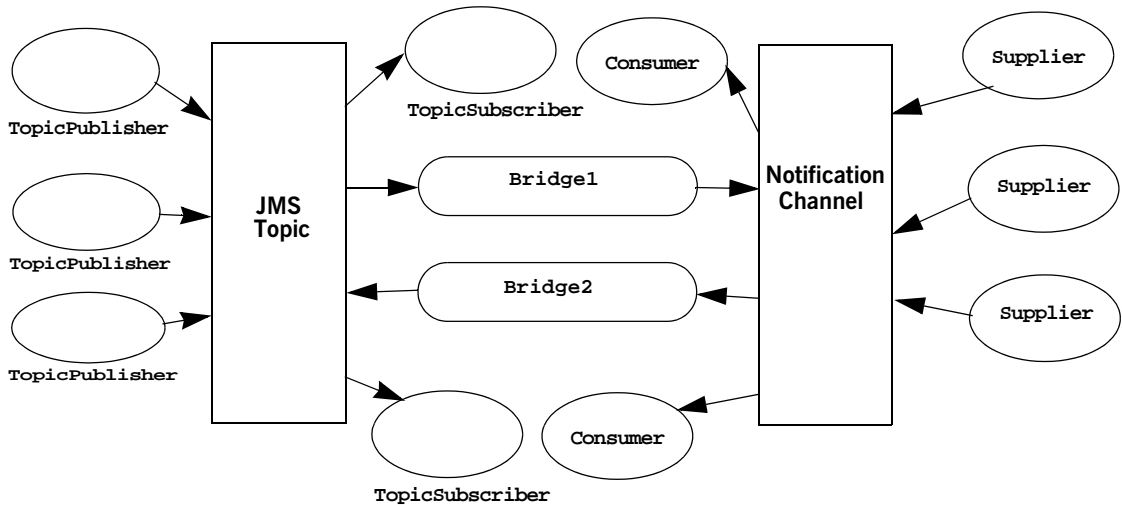
Figure 10: *JMS-Notification Bridging*

`Bridge1` is a JMS subcriber to the topic and consumes the JMS messages. It then converts them to a notification service event and pushes the events to the notification service, where the attached consumers can recieve them.

`Bridge2` is a notification push consumer attached to the notification service. When it recieves an event it converts it into a JMS message and publishes the message to the topic, where the subscribers can consume it.

Bridge endpoints

Bridges connect to JMS and the notification service using endpoints which mimic notification or JMS clients. For example, a bridge that passes messages from a JMS topic to a notification channel might have one

endpoint that acts like a durable JMS `TopicSubscriber` at the JMS side of the bridge and another endpoint that behaves like a `StructuredPushSupplier` on the notification service end of the bridge.

> **Note:** In fact, notification endpoints are specialized instances of proxy objects. In the example above the endpoint in the notification channel would appear as a bridge proxy consumer in the notification service console.

Message and property conversion The translation of message data and properties conforms to the OMG's *Notification / Java Message System Interworking* specification.

Essentially, JMS messages are translated into structured events with the JMS header and property data placed in the structured event header. Events are translated into JMS messages based on the following conventions:

- Any events are translated such that the data of the event is stored in the JMS message body and any QoS properties set for the message are placed in the appropriate fields of the JMS message header.
- Structured events are translated such that the data encapsulated in the message body is mapped to the JMS message body, the optional header fields and filterable date are mapped to user defined properties, and any QoS properties set in the header are mapped to the appropriate JMS header fields.
- Sequences of events are broken into single JMS messages according to the mapping for a structured event.

QoS service properties specifying the level of guarantee that a message is delivered, the lifetime of the message, and the priority of the message are preserved in the mapping and are enforced according to the specifications of each service.

For more information on the JMS-Notification bridge, see "The JMS-Notification Bridge Service" on page 277.

Part 2

The Notification Service

Developing Suppliers and Consumers

Client suppliers and consumers connect to an event channel in order to share information with each other.

Overview

The `CosNotifyComm` module defines client supplier and consumer interfaces. The interfaces can be categorized according to the following dependencies:

- A client interface supports either the push or pull model.
- For each push or pull model, an interface is defined to support one of the event message types: untyped, structured, or sequence.

The interface that you implement determines how a client sends or receives event messages.

In This Chapter

This chapter covers the following topics:

Obtaining an Event Channel

Overview

Client consumers and suppliers obtain an event channel object reference either by creating a channel, or by finding an existing one.

Procedure

You obtain an event channel by completing the following steps:

Step	Action
1	Obtain an event channel factory by calling `resolve_initial_references("NotificationService")`.
2	Use the event channel factory to create a channel or find an existing one.

Event Channel Factory Operations

You can call one of several operations on an event channel factory to create or find an event channel. By providing both create and find operations, the notification service allows any client or supplier to create an event channel, which other clients and suppliers can subsequently discover.

Orbix Notification supports two sets of event channel factory operations:

- The OMG-defined `CosNotifyChannelAdmin::EventChannelFactory` interface relies on system-generated IDs.
- Proprietary extensions in the `IT_NotifyChannelAdmin:: EventChannelFactory` interface allow user-defined channel names.

<table>
<tr>
<td>OMG Operations</td>
<td>

`CosNotifyChannelAdmin::EventChannelFactory` defines the following operations for obtaining an event channel:

</td>
</tr>
</table>

```
// IDL module CosNotifyChannelAdmin
interface EventChannelFactory {
  EventChannel create_channel(
                in CosNotification::QoSProperties initial_qos,
                in CosNotification::AdminProperties initial_admin,
                out ChannelID id)
  raises(CosNotification::UnsupportedQoS,
   CosNotification::UnsupportedAdmin);

  ChannelIDSeq get_all_channels();

  EventChannel get_event_channel(in ChannelID id)
      raises(ChannelNotFound);
};
```

create_channel() creates an event channel and returns an object reference.

get_all_channels() returns a sequence IDs of all event channels.

get_event_channel() returns an object reference to the ID-specified event channel.

Orbix Extensions

Orbix Notification provides proprietary operations for obtaining named event channels, in `IT_NotifyChannelAdmin::EventChannelFactory`:

```
// IDL module IT_NotifyChannelAdmin
struct EventChannelInfo
{
  string name;
  CosNotifyChannelAdmin::ChannelID id;
  CosNotifyChannelAdmin::EventChannel reference;
};

typedef sequence<EventChannelInfo> EventChannelInfoList;

// ...
interface EventChannelFactory :
    CosNotifyChannelAdmin::EventChannelFactory
```

```
{
// ...
 CosNotifyChannelAdmin::EventChannel create_named_channel(
        in  string name,
        in  CosNotification::QoSProperties initial_qos,
        in  CosNotification::AdminProperties initial_admin,
        out CosNotifyChannelAdmin::ChannelID id)
  raises(ChannelAlreadyExists, CosNotification::UnsupportedQoS,
         CosNotification::UnsupportedAdmin);

  CosNotifyChannelAdmin::EventChannel find_channel(
                          in string name,
                          out CosNotifyChannelAdmin::ChannelID id)
  raises(CosNotifyChannelAdmin::ChannelNotFound);
  CosNotifyChannelAdmin::EventChannel find_channel_by_id(
                          in CosNotifyChannelAdmin::ChannelID id,
                          out string name)
  raises(CosNotifyChannelAdmin::ChannelNotFound);
  // ...
  EventChannelInfoList list_channels();
};
```

create_named_channel() creates a named event channel and returns an object reference.

find_channel() returns an object reference to the named event channel.

find_channel_by_id() returns an object reference to an event channel based on the channel's ID.

list_channels() returns a list of event channels, which provides their names, IDs, and object references.

Example

The following code can be used by any supplier or consumer to obtain an event channel.

Example 1: *Obtaining an Event Channel*

```java
// Java
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
import org.omg.CosNotification.*;
import org.omg.CosNotifyChannelAdmin.*;

//Iona specific classes
import com.iona.corba.IT_NotifyChannelAdmin.*;

EventChannel ec = null;
EventChannelFactory m_factory = null;
IntHolder id = new IntHolder();
Property[] init_qos = new Property[0];
Property[] init_admin = new Property[0];
```

1
```java
Object obj =
  orb.resolve_initial_references("NotificationService");
m_factory = EventChannelFactoryHelper.narrow(obj);
```

2
```java
try ec = m_factory.create_named_channel("EventChannel",
                                        init_qos, init_admin, id)
```
3
```java
catch (ChannelAlreadyExists cae)
/* Channel already exists, so try to find it */
```
4
```java
  try
    {
      ec = m_factory.find_channel("EventChannel", id);
    }
  catch (ChannelNotFound cnf)
    {
      System.err.println("Could not create or find event
  channel");
      System.exit(1);
    }
  catch (SystemException sys)
    {
      System.err.println("System exception occurred during
                   find_channel: " +
                SystemExceptionDisplayHelper.toString(sys));
      System.exit(1);
    }
```

This code executes as follows:

1. Obtains the event channel factory.

2. Tries to create an event channel by calling `create_named_channel()`.

3. Catches the `IT_NotifyChannelAdmin::ChannelAlreadyExists` exception if a channel of the specified name already exists.

4. Tries to obtain an existing channel of the same name by calling `find_channel()`.

Implementing a Supplier

Actions

A client supplier program performs the following actions:

1. Instantiates suppliers using the appropriate interface in module `CosNotifyComm`.
2. Connects suppliers to the event channel.
3. Creates event messages.
4. Sends event messages to the event channel.
5. Disconnects from the event channel.

In This Section

This section discusses the following topics:

Instantiating the Supplier

Which Interface to Use?

Two dependencies determine which interface you should use to instantiate a supplier:

- The model that the supplier supports: push or pull.
- The type of event messages that the supplier generates: untyped, structured, or sequence of structures.

The IDL module `CosNotifyComm` defines six interfaces that support different combinations of both dependencies:

Event type	Push model	Pull model
untyped	`PushSupplier`	`PullSupplier`
structured	`StructuredPushSupplier`	`StructuredPullSupplier`
sequence	`SequencePushSupplier`	`SequencePullSupplier`

Example

You instantiate a supplier from the interface that supports the desired model and event message type. Example 2 shows how a client application might instantiate a supplier of type `StructuredPushSupplier`.

Example 2: *Instantiating a StructuredPushSupplier (Sheet 1 of 2)*

```
// Java
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
import org.omg.CosNotification.*;
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.CosNotifyComm.*;
import org.omg.TimeBase.*;

import org.omg.PortableServer.*;
import org.omg.PortableServer.POAPackage.*;
class NotifyPushSupplier extends StructuredPushSupplierPOA
  {
// Member variables not shown . . .
```

Example 2: *Instantiating a StructuredPushSupplier (Sheet 2 of 2)*

```
// The main entry point @param args command line args
    public static void main (String args[])
        {
// ORB and POA Activation not shown
// ...
        NotifyPushSupplier supplier = new NotifyPushSupplier();
// ...
    }
  public void NotifyPushSupplier()
    {
  // Implementation not shown ...
    }
}
```

Connecting to a Channel

Overview

In order to pass messages to the event channel, a supplier must connect to it through a proxy consumer that receives unfiltered events from the supplier. Each supplier must have its own proxy consumer. The proxy consumer begins the filtering process and passes the events down the channel.

Procedure

A client supplier connects to the event channel in three steps:

Step	Action
1	Obtain a `SupplierAdmin` object from the event channel.
2	Create a proxy consumer in the event channel, to receive the events that the supplier generates.
3	Connect to the proxy consumer.

Obtaining a Supplier Admin

On creation, an event channel instantiates a default `SupplierAdmin` object, which you obtain by calling `default_supplier_admin()` on the event channel. For example:

```
org.omg.CosNotifyChannelAdmin.SupplierAdmin sa =
  channels.default_supplier_admin();
```

The `EventChannel` interface also defines operations for creating and getting other supplier admin objects:

new_for_suppliers() returns a new supplier admin and its system-assigned `AdminID` identifier. When you create a supplier admin, you also determine whether to AND or OR its filters with proxy consumer filters (see).

get_supplieradmin() takes an `AdminID` identifier and returns an existing supplier admin.

get_all_supplieradmins() returns a sequence of `AdminID` identifiers.

Why Create Multiple Admin Objects?

You might want to create multiple supplier admin objects for one of the following reasons:

- Groups of proxy consumers each require the same quality-of-service properties. All proxy consumers inherit properties from their parent supplier admin. By creating different supplier admin objects with the desired sets of properties, you can more easily manage the properties of individual proxies.

  For more information about quality-of-service properties, see Chapter 5.

- Groups of proxy consumers have different filtering requirements. You can set different filters on individual admin objects and group proxy consumers accordingly.

- You need to distribute the load of event messages among different supplier admin objects. A supplier admin's workload is liable to increase for two reasons: using supplier-side forwarding filters (see "Forwarding Filters" on page 84), and implementing pull-model suppliers. One or both factors might require additional supplier admin objects to handle the extra work load that these entail.

Proxy Consumers

A proxy consumer is responsible for receiving event messages from its client supplier and inserting them into the event channel, where they are forwarded to all interested consumers. You create one proxy consumer for each client supplier.

As with client suppliers, you can create six types of proxy consumers, depending on the client supplier's model (push/pull) and event message type (untyped, structured, or sequence of structures). The type of proxy consumer must match the type of its client supplier.

The `CosNotifyChannelAdmin` module defines interfaces that support the following proxy consumer objects:

```
ProxyPushConsumer
StructuredProxyPushConsumer
SequenceProxyPushConsumer
ProxyPullConsumer
StructuredProxyPullConsumer
SequenceProxyPullConsumer
```

Obtaining a Proxy Consumer

You obtain a proxy consumer by invoking one of the following operations on a supplier admin:

obtain_notification_push_consumer() returns a push-model proxy consumer.

obtain_notification_pull_consumer() returns a pull-model proxy consumer.

Both methods take one of the following arguments, which determines the event message type that this proxy consumer handles:

```
ANY_EVENT
STRUCTURED_EVENT
SEQUENCE_EVENT
```

Both methods raise `CosNotifyChannelAdmin::AdminLimitExceeded` when the event channel's `MaxSuppliers`(see "MaxSuppliers" on page 81) limit is reached.

Example

The code in Example 3 obtains a `StructuredProxyPushConsumer` proxy consumer for a `StructuredPushSupplier` supplier by calling `obtain_notification_push_consumer()`, and supplying an argument of `STRUCTURED_EVENT`.

Example 3: *Obtaining a Proxy Consumer*

```java
// Java
import org.omg.CosNotifyChannelAdmin.*;

IntHolder proxy_id = new IntHolder();
ClientType ctype = ClientType.STRUCTURED_EVENT;
try
{
  ProxyConsumer obj =
    sa.obtain_notification_push_consumer(ctype, proxy_id);
}
catch(AdminLimitExceeded err)
{
  // handle the exception
}

StructuredProxyPushConsumer ppc =
  StructuredProxyPushConsumerHelper.narrow(obj);
```

Connecting a Supplier to a Proxy Consumer

After creating a proxy consumer, you can connect it to a compatible client supplier. This establishes the client supplier's connection to the event channel, so it can send messages.

Each proxy consumer interface supports a connect operation; the operation requires that the supplier and its proxy support the same delivery model and event-message type. For example, the `StructuredProxyPushConsumer` interface defines `connect_structured_push_supplier()`, which only accepts an object reference to a `StructuredPushSupplier` as input.:

```
// IDL
interface StructuredProxyPushConsumer :
    ProxyConsumer, CosNotifyComm::StructuredPushConsumer
{
 void connect_structured_push_supplier(
        in CosNotifyComm::StructuredPushSupplier push_supplier)
  raises(CosEventChannelAdmin::AlreadyConnected);
};
```

Example

Example 4 shows one method of implementing a `StructuredPushSupplier` client that connects itself to a proxy consumer.

Example 4: *Connecting a StructuredPushSupplier (Sheet 1 of 2)*

```
// Java
import org.omg.CosEventChannelAdmin.*;

class NotifyPushSupplier extends StructuredPushSupplierPOA
  {
// ...
    public static void main (String args[])
      {
// ORB and POA creation not shown
// proxy ppc and PushSupplier supplier obtained previously
        try
          {
            ppc.connect_structured_push_supplier(supplier);
          }
        catch (AlreadyConnected.value ac)
          {
          // Handle the exception
          }
```

Example 4: *Connecting a StructuredPushSupplier (Sheet 2 of 2)*

```
       catch (SystemException sys)
         {
          System.err.println("Encountered system exception
                     during connect: " +
                   SystemExceptionDisplayHelper.toString(sys));
          System.exit(1);
          }
     // ...
     }
  }
```

Creating Event Messages

Types of Event Messages

The notification service supports three formats for sending events:

- *Untyped* events are sent as CORBA::Any types. Clients can store an event message into any format they choose, including a structure, then package the data into an Any.
- *Structured* events provide a well-defined data structure that encapsulates an event's type and other information. Filters use this data to screen event messages.
- *Sequences* of structured events are simply batches of structured events gathered together and sent at the same time.

Structured Event Messages

Structured event messages are defined in module CosNotification as follows:

```
struct Property {
    PropertyName name;
    PropertyValue value;
};
typedef sequence<Property> PropertySeq;

typedef PropertySeq OptionalHeaderFields;
typedef PropertySeq FilterableEventBody;

struct EventType {
    string domain_name;
    string type_name;
};

struct FixedEventHeader {
    EventType event_type;
    string event_name;
};

struct EventHeader {
    FixedEventHeader fixed_header;
    OptionalHeaderFields variable_header;
};
```

```
struct StructuredEvent {
    EventHeader header;
    FilterableEventBody filterable_data;
    any remainder_of_body;
};
```

Each structured event has three main components, as shown in Figure 11.
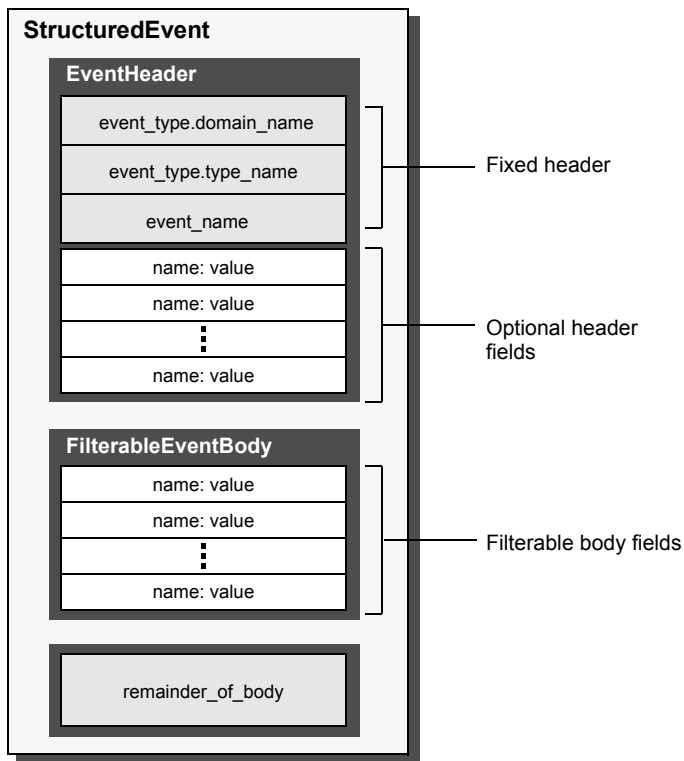
Figure 11: *Structured Event Components*

EventHeader consists of two members:

- A fixed header section that contains three string fields for specifying event-type data: domain_name, type_name, and event_name.

- A list of zero or more optional header fields. Each field name is a string, and each value is a CORBA::Any. These fields are typically used to set properties on an event message, such as its lifetime and priority.

FilterableEventBody consists of data fields that can be used to set user-defined properties. Filters typically use these to screen event messages.

remainder_of_body is a CORBA::Any, which can store any event-related data, such as the contents of a file.

Why Use Structured Event Messages?

A structured event message can provide filterable information, such as the event's type and contents, and assign quality-of-service properties to the event, such as its priority or lifetime. Later chapters in this guide describe notification filters (Chapter 6) and quality-of-service properties (Chapter 5).

Example

The code in Example 5 shows how a supplier creates a structured message that sets an event type's domain name and type name to SportsNews and BaseballResults, respectively, and sets its priority to 0.

Example 5: *Creating a Structured Message*

```java
// Java
import org.omg.CosNotification.*;
```
1
```java
StructuredEvent event = new StructuredEvent();
```

2
```java
String domain_name = new String("SportsNews");
String type_name = new String("BaseballResults");
EventType event_type = new EventType(domain_name, type_name);
```
3
```java
String event_name = new String("");
FixedEventHeader fixed_header = new FixedEventHeader(event_type,
                                                     event_name);
```
4
```java
String property_name = new String(Priority.value)
Property[] variable_header = new Property[1];
variable_header[0] = new Property();
variable_header[0].name = property_name;
variable_header[0].value = orb.create_any();
variable_header[0].value.insert_long(0);
```

5
```java
event.header = new EventHeader(fixed_header, variable_header);
```

6
```java
event.filterable_data = new Property [0];
event.remainder_of_body = ORB.create_any();
```

This code executes as follows:

1. Creates an event.
2. Builds a new event header.
3. Builds a new fixed event header.
4. Builds a new property list in the variable header.
5. Adds the fixed header and the variable header to the event.
6. Creates the remainder of the event body.

Sending Event Messages

A client supplier sends event messages in one of two ways:

- A push supplier invokes the appropriate push operation on its proxy consumer and supplies the event as an input argument.
- A pull supplier implements the appropriate pull or try_pull operation. When the proxy consumer invokes one of these operations, the supplier returns an event message, if one is available.

Push Supplier

A push supplier invokes one of the following push operations on its proxy consumer, according to the event messages that they support:

- push() is invoked by a PushSupplier and accepts a CORBA::Any as input.
- push_structured_event() is invoked by a StructuredPushSupplier and accepts a StructuredEvent as input.
- push_structured_events() is invoked by a SequencePushSupplier and accepts a sequence of event structures as input.

Example

Example 6 pushes a structured event message.

Example 6: *Pushing a Structured Event*

```
// Java
// proxy consumer and event message already obtained
try
  {
    proxy.push_structured_event(se);
  }
catch (SystemException sys)
  {
    System.err.println("Unexpected system exception during push:"
                 +SystemExceptionDisplayHelper.toString(sys));
    System.exit(1);
  }
catch (org.omg.CosEventComm.Disconnected dc)
  {
    System.err.println("Channel is disconnected.");
    System.exit(1);
  }
```

Example 6: *Pushing a Structured Event*

```
catch (Exception e)
  {
    System.err.println("Unknown exception occurred during push");
    System.exit(1);
  }
```

Pull Supplier

A pull supplier sends event messages only on request. Depending on the setting of the configuration variable `dispatch_strategy`, a pull supplier's proxy consumer invokes a `try_pull()` or a `pull()` operation on it's supplier. Pull suppliers are responsible for implementing the appropriate variant of `try_pull()` or `pull()`. Each pull supplier interface supports a `try_pull()` and `pull()` operation:

- `try_pull()` and `pull()` are invoked on a `PullSupplier` and return a `CORBA::Any`.

- `try_pull_structured_event()` and `pull_structured_event()` are invoked on a `StructuredPullSupplier` and return a `CosNotification::StructuredEvent`.

- `try_pull_structured_events()` and `pull_structured_events()` are invoked on a `SequencePullSupplier` and return a sequence of event structures.

A `try_pull` operation is non-blocking and is called by the proxy when the notification service's `dispatch_strategy` is set to `thread_pool`. It returns immediately with an output parameter of type boolean to indicate whether the return value actually contains an event. The proxy consumer continues to invoke the pull operation on the supplier as many times as specified in the `MaxRetries` property (see "MaxRetries" on page 79). The interval between retries is specified by the `PullInterval` property (see "PullInterval" on page 80).

A `pull` operation is blocking and is called by the proxy when the notification service's `dispatch_strategy` is set to `single_thread`. It blocks until an event is ready to be forwarded to the proxy.

Since the setting of the notification service's `dispatch_strategy` cannot typically be determined at development time, the safest approach to developing pull style suppliers is implement both `try_pull()` and `pull()`.

Example

Example 7 implements `try_pull_structured_event()` by attempting to populate an event structure with the latest baseball scores.

Example 7: *Pulling Structured Events*

```java
// Java
import org.omg.CosNotification.*;

class NotifyPullSupplier extends StructuredPullSupplierPOA
  {
// ...
    public StructuredEvent try_pull_structured_event
   (BooleanHolder has_event)
      {
        StructuredEvent se = new StructuredEvent();
        has_event.value = false;

// get scores
        String scores;
        boolean has_scores = get_scores(scores);

// If there are scores build the event
        if (has_scores == true)
          {
            String domain_name = new String ("SportsNews");
            String type_name = new String ("BaseballResults");
            EventType event_type = new EventType(domain_name,
                                                 type_name);

            String event_name = new String("");
            FixedEventHeader fixed_header =
              new FixedEventHeader(event_type, event_name);

            Property[] variable_header = new Property[0];
            se.header = new EventHeader(fixed_header,
                                        variable_header);

            se.filterable_data = new Property [0];

            se.remainder_of_body = ORB.create_any();
            se.remainder_of_body.insert_string(scores);

            has_event.value = true;
          }
```

Example 7: *Pulling Structured Events*

```
    return se;
}
```

Disconnecting From the Event Channel

Overview

A client supplier can disconnect from the event channel at any time by invoking the disconnect operation on its proxy consumer. This operation terminates the connection between a supplier and its target proxy consumer. The channel then releases all resources allocated to support its connection to the supplier, including destruction of the target proxy consumer.

Each proxy consumer interface supports a disconnect operation. For example, `disconnect_structured_push_consumer()` is defined in the interface `StructuredProxyPushConsumer`.

Implementing a Consumer

Actions

A client consumer program performs the following actions:

1. Instantiates consumers using the appropriate `CosNotifyComm` interface.
2. Connects consumers to the event channel.
3. Obtains event messages.
4. Disconnects from the event channel.

In This Section

This section covers the following topics:

Instantiating a Consumer

Which Interface to Use?

Two dependencies determine which interface you use to instantiate a consumer:

- The model that the consumer supports: push or pull.
- The type of event messages that the consumer receives: untyped, structured, or sequence of structures.

The IDL module `CosNotifyComm` defines six interfaces that support different combinations of both dependencies:

Event type	Push model	Pull model
untyped	`PushConsumer`	`PullConsumer`
structured	`StructuredPushConsumer`	`StructuredPullConsumer`
sequence	`SequencePushConsumer`	`SequencePullConsumer`

You instantiate a consumer from the interface that supports the desired model and event message type.

Example

Example 8 shows how a client application might instantiate a structured push consumer.

Example 8: *Instantiating a Consumer (Sheet 1 of 2)*

```
// Java
import org.omg.CORBA.*;
import org.omg.CosNotification.*;
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.CosNotifyComm.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POAPackage.*;

class NotifyPushConsumer extends StructuredPushConsumerPOA
{
// member variables not shown...
```

Example 8: *Instantiating a Consumer (Sheet 2 of 2)*

```
// The main entry point @param args command line args
  public static void main (String args[])
    {
      // ORB and POA initialization not shown ...

      NotifyPushConsumer consumer = new NotifyPushConsumer();

      // ...
    }

  void public NotifyPushConsumer()
    {
    }

// ...
}
```

Connecting to the Channel

Overview

Consumers receive messages from the event channel through a proxy supplier. Each consumer on the channel has its own proxy supplier. Proxy suppliers use the same delivery method as their consumers and send the appropriate message type.

Procedure

Consumers connect to the event channel in three steps:

Step	Action
1	Obtain a ConsumerAdmin object from the event channel.
2	Create a proxy supplier in the event channel, to receive supplier-generated event messages.
3	Connect to the proxy supplier.

Obtaining a Consumer Admin

On creation, an event channel instantiates a default ConsumerAdmin object, which you supply by calling default_consumer_admin() on the event channel. For example:

```
org.omg.CosNotifyChannelAdmin.ConsumerAdmin ca =
  channel.default_consumer_admin();
```

The EventChannel interface also defines operations for creating and getting other consumer admin objects:

new_for_consumers() returns a new consumer admin and its system-assigned AdminID identifier. When you create a consumer admin, you also determine whether to AND or OR its forwarding filters with proxy supplier filters (see "Traversing Multiple Filters in a Channel" on page 90).

get_consumeradmin() takes an AdminID identifier and returns an existing consumer admin.

get_all_consumeradmins() returns a sequence of AdminID identifiers.

Why Create Multiple Admin Objects?

You might want to create multiple consumer admin objects for one of the following reasons:

- Groups of proxy suppliers each require the same quality-of-service properties. All proxy suppliers inherit properties from their parent consumer admin. By creating different consumer admin objects with the desired sets of properties, you can more easily manage the properties of individual proxies.

  For more information about quality-of-service properties, see Chapter 5.

- Groups of proxy suppliers each have the same filtering requirements. Because all event messages are initially filtered by the consumer admin, you can use admin filters to centralize filter processing and administration, and minimize the associated overhead.

- You need to distribute the load of event messages among different consumer admin objects. A consumer admin's work load is liable to increase for two reasons: using consumer-side filters, and the number of message-forwarding proxies. One or both factors might require additional consumer admin objects to handle the extra work load that these entail.

  For more information about filters, see Chapter 6.

Proxy Suppliers

A proxy supplier is responsible for distributing event messages that have been sent by the event channel to its consumer, subject to filtering and quality-of-service settings. You create one proxy supplier for each client consumer.

As with client consumers, you can create six types of proxy suppliers, depending on the client consumer's model (push/pull) and event message type (untyped, structured, or sequence of structures). The proxy supplier must be the same type as its client consumer.

The module `CosNotifyChannelAdmin` defines interfaces that support the following proxy supplier objects:

```
ProxyPushSupplier
StructuredProxyPushSupplier
SequenceProxyPushSupplier
ProxyPullSupplier
StructuredProxyPullSupplier
```

SequenceProxyPullSupplier

Obtaining a Proxy Supplier   You obtain a proxy supplier by invoking one of the following methods on a consumer admin:

obtain_notification_push_supplier() returns a push-model proxy supplier.

obtain_notification_pull_supplier() returns a pull-model proxy supplier.

Both methods take one of the following arguments, which determines the event message type that this proxy supplier handles:

```
ANY_EVENT
STRUCTURED_EVENT
SEQUENCE_EVENT
```

Both methods raise CosNotifyChannelAdmin::AdminLimitExceeded when the event channel's MaxConsumers(see "MaxConsumers" on page 81) limit is reached.

Example   Example 9 obtains a proxy supplier for a StructuredPushConsumer supplier by calling obtain_notification_push_supplier().

Example 9: *Obtaining a Proxy Supplier*

```java
// Java
import org.omg.CosNotifyChannelAdmin.*;

IntHolder proxy_id = new IntHolder();
ClientType ctype = ClientType.STRUCTURED_EVENT;

try
{
  ProxySupplier obj =
    ca.obtain_notification_push_supplier(ctype, proxy_id);
}
catch(AdminLimitExceeded err)
{
  //handle exception
}

StructuredProxyPushSupplier pps =
  StructuredProxyPushSupplierHelper.narrow(obj);
```

Connecting a Consumer to a Proxy Supplier

After creating a proxy supplier, you can connect it to a compatible client consumer. This establishes the client's connection to the event channel, so it can obtain messages from suppliers.

Each proxy supplier interface supports a connect operation; the operation requires that the client supplier and its proxy support the same push or pull model and event-message type. For example, the `StructuredProxyPushSupplier` interface defines `connect_structured_push_consumer()`, which only accepts an object reference to a `StructuredPushSupplier` as input:

```
// IDL
interface StructuredProxyPushSupplier :
    ProxySupplier,
    CosNotifyComm::StructuredPushSupplier
{
  void connect_structured_push_consumer
      (in CosNotifyComm::StructuredPushConsumer push_consumer)
  raises(CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError);
};
```

Example

Example 10 shows how you might implement a `StructuredPushConsumer` client that connects itself to a proxy supplier.

Example 10: *Connecting to a Proxy Supplier (Sheet 1 of 2)*

```
// Java
import org.omg.CosNotifyChannelAdmin.*;

class NotifyPushConsumer extends StructuredPushConsumerPOA
{
// ...

  public static void main (String args[])
    {
// ...
// Proxy pps and PushConsumer consumer obtained previously
      try
        {
          pps.connect_structured_push_consumer(consumer);
        }
```

Example 10: *Connecting to a Proxy Supplier (Sheet 2 of 2)*

```
      catch (AlreadyConnected.value ac)
        {
          System.err.println("Already connecting to channel.");
          System.exit (1);
        }
      catch (SystemException sys)
        {
        System.err.println("Encountered system exception during
        connect: " + SystemExceptionDisplayHelper.toString(sys));
        System.exit(1);
        }

//...
    }
}
```

Obtaining Event Messages

Overview

A client consumer obtains event messages in one of two ways:

- A push consumer implements the appropriate `push` operation. As events become available, the proxy supplier pushes them to its client consumer in the appropriate format.

- A pull consumer invokes the appropriate `pull` or `try_pull` operation on its proxy supplier; the proxy supplier returns with the next available event.

Event Message Conversion

If necessary, the event channel converts event messages to the type expected by its consumers. For example, if a `PushSupplier` pushes an untyped event message to an event channel that has `StructuredPushConsumer` clients, the channel delivers the event to those clients as a structured event message. The event data is stored in the message's `remainder_of_body` member. Similarly, `PushConsumer` clients receive an event originally sent in structured format as a `CORBA::Any`.

Push Consumer

A push consumer implements one of the following push operations:

- `push()` is implemented by a `PushConsumer`, and receives an event message of the `CORBA::Any` type.

- `push_structured_event()` is implemented by a `StructuredPushConsumer` and receives an event message of `CosNotification::StructuredEvent`.

- `push_structured_events()` is implemented by a `SequencePushConsumer` and receives a sequence of structured event messages `CosNotification::EventBatch`.

Example

Example 11 implements push_structured_event() to receive a structured event that contains sports scores.

Example 11: *Receiving Events Using Push*

```java
// Java
import org.omg.CosNotification.*;

class NotifyPushConsumer extends StructuredPushConsumerPOA
{
// ...

  public void push_structured_event(StructuredEvent event)
    {
      String news_type = new
       String(event.header.fixed_header.event_type.domain_name);
      String sports_type = new
       String(event.header.fixed_header.event_type.type_name);

       if(news_type.equals("SportsNews"))
         {
           String scores =
             event.remainder_of_body.extract_string();

           System.out.println("Current " + sports_type + "scores:
                              " + scores);
         }
    }

//...
}
```

Pull Consumer

A pull client consumer invokes the appropriate pull or try_pull operation on its proxy supplier to solicit event messages; the proxy supplier returns with the next available event.

Each proxy supplier interface supports a variant of the pull and the try_pull operations:

- pull() and try_pull() are invoked on a PullSupplier proxy and return a CORBA::Any argument.

- pull_structured_event() and try_pull_structured_event() are invoked on a StructuredPullSupplier proxy and return a CosNotification::StructuredEvent.
- pull_structured_events() and try_pull_structured_events() are invoked on a SequencePullSupplier proxy and return a sequence of event structures.

The pull and try_pull operations differ only in their blocking mode:

- A pull operation blocks until an event is available.
- A try_pull operation is non-blocking—it returns immediately with a boolean output parameter to indicate whether the return value actually contains an event. The proxy consumer continues to invoke the pull operation on the supplier as many times as specified in the MaxProxyConsumerRetries property (see "MaxRetries" on page 79). The interval between retries is specified by the PullInterval property (see "PullInterval" on page 80).

Example

Example 12 shows how one might use try_pull to receive data from a StructuredProxyPullSupplier.

Example 12: *Pulling Events (Sheet 1 of 2)*

```java
// Java
BooleanHolder has_data = new BooleanHolder();

try
  {
    event = proxy.try_pull_structured_event(has_data);
  }
catch (org.omg.CosEventComm.Disconnected dsc )
  {
    System.err.println("Disconnected exception occured during
                       pull");
     System.exit (1);
   }
catch (SystemException sys )
  {
    System.err.println("System exception occured during pull");
    System.exit (1);
  }
```

Example 12: *Pulling Events (Sheet 2 of 2)*

```
if (has_data.value)
  {
    n = event2.remainder_of_body.extract_ulong ();
    System.out.println("Received event number " + n +  " using
                           try pull");
  }
```

Disconnecting From the Event Channel

Overview

A client consumer can disconnect from the event channel at any time by invoking the disconnect operation on its proxy supplier. This operation terminates the connection between the consumer and its target proxy supplier. The event channel then releases all resources allocated to support its connection to the consumer, including destruction of the target proxy supplier.

Each proxy supplier interface supports a disconnect operation. For example, `disconnect_structured_push_supplier()` is defined in `StructuredProxyPushSupplier`.

Notification Service Properties

You can set and modify a number of properties on notification service components.

Overview

Notification service properties control the delivery of event messages—for example, their priority and reliability. You can use either the API or the Notification Console to set these properties on a channel, an administration object, a proxy object, or an event message.

In This Chapter

This chapter covers the following topics:

Property Types

In This Section

This section covers the following topics:

Administration Properties

Administration properties control the behavior of event channels and cannot be set on other objects. They are supported by the `AdminPropertiesAdmin` interface, which provides the accessor operations `get_admin()` and `set_admin()`.

The notification service supports the following administration properties:

```
MaxConsumers
MaxSuppliers
MaxQueueLength
RejectNewEvents
```

Quality-of-Service Properties

Quality-of-service properties control the behavior of all notification service components and can be set on any notification service object, including messages. They are supported by the `QoSAdmin` interface, which provides accessor operations `get_qos()` and `set_qos()`.

Table 2 lists the quality-of-service properties and the component types on which they can be set. Some properties have more specific restrictions; these are discussed in the property descriptions (see "Property Descriptions" on page 69).

Table 2:   *Component Support for Quality-of-Service Properties*

Property	Message	Proxy	Admin	Channel
EventReliability	Y			Y
ConnectionReliability		Y	Y	Y
Priority	Y	Y	Y	Y

Table 2: *Component Support for Quality-of-Service Properties*

Property	Message	Proxy	Admin	Channel
OrderPolicy		Y	Y	Y
StopTime	Y			
StopTimeSupported		Y	Y	Y
Timeout	Y	Y	Y	Y
StartTime	Y			
StartTimeSupported		Y	Y	Y
MaxEventsPerConsumer		Y	Y	Y
DiscardPolicy		Y	Y	Y
MaximumBatchSize		Y	Y	Y
PacingInterval		Y	Y	Y
MaxRetries		Y	Y	Y
RetryTimeout		Y	Y	Y
MaxRetryTimeout		Y	Y	Y
RequestTimeout		Y	Y	Y
PullInterval		Y	Y	Y
RetryMultiplier		Y	Y	Y

Property Inheritance

Order of Inheritance

On creation, an event channel, admin, or proxy initially inherits its quality-of-service properties from the following components, in ascending order of precedence

- The notification service's default property settings.
- Component ancestors, in order of creation.

For example, when you create a consumer proxy, the notification service:

1. Obtains its own default properties

2. Merges these properties with notification channel properties.

3. Merges the aggregate of all higher-level properties with the parent supplier admin's properties

4. Sets the merged list of properties on the consumer proxy.

At each merge stage, the current object's properties override corresponding properties of all higher-level components.

WARNING: If you change a component's properties, the changes are inherited only by child components that are created afterwards; existing child components are unaffected by changes in their parents.

Setting Properties

Overview

Properties can be set on the following notification service components, in ascending order of precedence:

- Event channel
- Admins
- Proxies
- Structured event messages

Properties can be set programmatically or through the Notification Console. Properties can also be set for individual structured events through their optional header fields.

Consistency

Because properties can be set individually on the different components that handle event message delivery, it is important to ensure consistent settings across the entire delivery path. Unless all of the components in the delivery path agree on a consistent set of policies, message delivery can be unpredictable.

In This Section

This section covers the following topics:

Setting Properties Programmatically

Methods for Setting Properties

The notification service provides two methods for setting an object's properties:

- `set_admin()` sets administration properties on an event channel. It cannot be used to set properties on other notification service objects.
- `set_qos()` sets quality-of-service properties on all notification service objects.

set_admin()

`set_admin()` is called on an event channel to set one of the following administration properties:

```
MaxConsumers
MaxSuppliers
```

You can use `set_admin()` to change existing properties on an event channel or set new ones. Any property that is not specified remains unchanged.

`set_admin()` takes a single argument of type `CosNotification::AdminProperties`, which is defined as a sequence of `String`/`Any` name-value pairs specifying the properties to be changed and their new settings.

`set_admin()` throws an exception of `UnsupportedAdmin` if the property is unsupported for the target component. This exception returns a sequence of structures containing the name of the invalid property, an error code identifying the error, and a `cstruct` specifing the valid range of settings for the property.

Table 3 lists the possible error codes returned because of an `UnsupportedAdmin` exception.

set_qos()

`set_qos()` can be called on all notification service components to set their quality-of-service properties.

You can use `set_qos()` to change existing properties on any notification service component or to set new ones. Any property that is not specified remains unchanged.

`set_qos()` takes a single argument of type
`CosNotification::QoSProperties` which is defined as a sequence of
`String/Any` name-value pairs specifying the properties to be changed and
their new settings.

`set_qos()` can throw `UnsupportedQoS`, if the property is unsupported for the
target component. This exception returns a sequence of structures
containing the name of the invalid property, an error code identifying the
error, and a `cstruct` specifying the valid range of settings for the property.

Table 3 lists the possible error codes returned because of an
`UnsupportedQoS` exception.

Table 3:   *Error Codes returned with the UnsupportedQoS and
UnsupportedAdmin Exceptions*

Error code	Meaning
UNSUPPORTED_PROPERTY	Orbix does not support the property for this type of object.
UNAVAILABLE_PROPERTY	This property cannot be combined with existing quality-of-service properties.
UNSUPPORTED_VALUE	The value specified for this property is invalid for the target object. A range of valid values is returned.
UNAVAILABLE_VALUE	The value requested for this property is invalid in the context of other quality-of-service properties currently in force. A range of valid values is returned.
BAD_PROPERTY	The property name is unknown.
BAD_TYPE	The type supplied for the value of this property is incorrect.
BAD_VALUE	The value supplied for this property is illegal. A range of valid values is returned.

Example

Example 13 shows one way to set an event channel's `OrderPolicy` to
`FifoOrder`.

Example 13: *Setting Qos Properties (Sheet 1 of 2)*

```
\\ Java
\\ Event channel chan obtained earlier
import org.omg.CosNotification;
```

Example 13: *Setting Qos Properties (Sheet 2 of 2)*

```
try
  {
    Property[] NewQoS = Property[1];
    NewQoS[0] = new Property();
    NewQoS[0].name = OrderPolicy.value;
    NewQos[0].value = ORB.init().create_any();
    NewQos[0].value.insert_short(FifoOrder.value);

    chan.set_qos(qos);
  }
catch (org.omg.CosNotification.UnsupportedQoS uqos)
  {
    System.exit(1);
    System.err.println("UnsupportedQoS Exception");
  }
```

Setting a Structured Event's QoS Properties

Overview

You can set quality-of-service properties in a structured event message's header. These settings override the corresponding properties specified for the consumer and supplier proxies; however, they apply only to that event.

BAD_QOS Exception

If the requested property is invalid, the notification service raises system exception BAD_QOS. This exception is thrown during transmission of a structured event from a supplier to the channel when the channel determines that it cannot accept the event header properties.

The BAD_QOS exception provides no details about why it was thrown. By calling validate_event_qos() in advance, a client can verify whether it can safely set a property in an event message header. For more on this operation .

Example

sets a structured event's Priority property to 0.

Example 14: *Setting QoS Properties in an Event Header*

```java
// Java
import org.omg.CosNotification;

StructuredEvent event = new StructuredEvent();

event.header = new EventHeader();

event.header.fixed_header = new FixedEventHeader();
event.header.fixed_header.event_type = new
  EventType("SportNews", "BaseballResults");
event.header.fixed_header.event_name = new String("");

event.header.variable_header = new Property[1];
event.header.variable_header.name = Priority;
event.header.variable_header.value = ORB.create_any();
event.header.variable_header.value.insert_short(0);

event.filterable_data = new Property [0];
```

Getting Properties

Methods

The notification service provides methods for looking at a notification service object's properties. Depending on a property's type (see "Property Types" on page 58), you can call either `get_admin()` or `get_qos()` on a notification service object to retrieve its properties.

get_admin()

`get_admin()` takes no input parameters, and returns a sequence of `CosNotification::AdminProperties` which contains name-value pairs encapsulating the current administrative settings for the target channel.

get_qos()

`get_qos()` retrieves the effective quality-of-service properties for a channel, admin, or proxy. It returns the list of properties, and their values, that are set on the target object, including those properties inherited from higher levels, in a sequence of name-value pairs of type `CosNotification::QoSProperties`.

Example

Example 15 gets the quality-of-service properties that are set for channel `chan`.

Example 15: *Getting QoS Properties*

```
// Java
org.omg.CosNotification.Property[] current_qos;

current_qos = chan.get_qos();
```

Validating Properties

Methods

The notification service supports two methods that lets a supplier check whether a given object supports one or more quality-of-service properties:

- `validate_qos()` can be called on all notification service objects.
- `validate_event_qos()` can only be called on consumer proxies to determines which quality-of-service properties are valid for an event message.

Parameters

Both methods take an input and output parameter:

required_qos: A sequence of quality-of-service property name-value pairs of type `CosNotification::QoSProperties` that specify a set of quality-of-service settings.

available_qos: An output parameter that contains a sequence of `CosNotification::PropertyRange` data structures. Each element in this sequence includes the name of an additional quality-of-service property supported by the target object that could have been included on the input list and resulted in a successful return from the operation, along with the range of values that would have been acceptable for each such property.

`available_qos` only returns properties that have no interdependencies. If two properties are interdependent—for example, `EventReliability` and `ConnectionReliability`—then neither is returned.

UnsupportedQoS Exception

If any of the properties listed in `required_qos` are invalid for the target object, the call throws an `UnsupportedQoS` exception, which shows which properties are invalid and why. For more information on return codes, see Table 3 on page 63.

Example

In Example 16, a supplier calls `validate_event_qos()` on the proxy consumer `ppc` to determine whether it can accept a structured event whose `EventReliability` property is set to `Persistent`.

Example 16: *Validating Event Properties*

```java
// Java
// consumer proxy ppc obtianed earlier
import org.omg.CosNotification;

Property[] QoS = new Property[1];
QoS[0] = new Property();
QoS[0].name = new String(EventReliability.value);
Qos[0].value = ORB.create_any();
Qos[0].value.insert_short(Persistent.value);

try
  {
    ppc.validate_event_qos(QoS);
  }
catch(UnsupportedQoS unsupported)
  {
    System.err.prntln("Event persistence not allowed. Error:
                       unsupported.");
  }
catch(org.omg.CORBA.SystemException se)
  {
    System.err.prntln("System exception occurred during
                       validate_event_qos call.");
  }
```

Property Descriptions

Topics

The following topics are discussed in this section:

Reliability Properties

Property Names

The notification service defines two reliability properties that determine how it handles service fail over:

- EventReliability
- ConnectionReliability

EventReliability

`EventReliability` specifies level of assurance that an event will be delivered over multiple restarts of the process hosting its event channel. This property can be set on an event channel and on individual events. By default, an event's reliability is set to match the event channel.

You can set this property to `BestEffort` or `Persistent`:

BestEffort: (default) A queued event remains viable only during the event channel's hosting process' lifetime. If the event channel's hosting process fails, delivery cannot be guaranteed for any buffered best-effort events; and consumers might receive the same event more than once.

Persistent: A queued event is persistent. If the event channel's hosting process fails, all persistent events that remain within their expiry limits are restored when the channel's hosting process is restarted.

Note: `EventReliability` on a per event basis is only effective when the channel's `EventReliability` is set to `Persistent`. Otherwise, all events will be delivered with `BestEffort`.

ConnectionReliability

`ConnectionReliability` specifies whether a channel maintains information about connected suppliers and consumers beyond its hosting processes current lifetime. This property can be set only on a channel.

You can set this property to `BestEffort` or `Persistent`:

BestEffort: (default) Supplier and consumer connections are valid only during the event channel's hosting process' current lifetime. If the event channel's hosting process fails, all references to that event channel become

invalid and should be explicitly disconnected by the consumers and suppliers. Upon restart of the channel's hosting process, all suppliers and consumers must reconnect to the channel using new references.

Persistent: All supplier and consumer connections remain viable beyond the event channel's hosting process' current lifetime. Upon restart from a failure, the event channel automatically re-establishes connections to all clients that were connected to it at the time of failure.

Valid Combinations

The following matrix shows which combinations for `EventReliability` and `ConnectionReliability` are valid:

EventReliability	ConnectionReliability	
	BestEffort	Persistent
BestEffort	Y	Y
Persistent	N	Y

Event Queue Order

OrderPolicy

The `OrderPolicy` property tells a proxy in what order to queue events for delivery. This property can be set on a channel, and on individual admin or proxy objects; it is typically set by a consumer on its consumer admin, supplier proxy, or both.

Values

You set this policy with one of the following constants:

AnyOrder: Queue events in any order. In practice, this has the same effect as specifying `FifoOrder`.

FifoOrder: Queue events in the order they are received by the event channel.

PriorityOrder: (default) Queue events according to their `Priority` property setting, so higher priority events are delivered before lower priority events.

DeadlineOrder: Queue events in order of expiry deadlines, so events that are destined to expire earliest are delivered first.

Event Priority

Priority

The `Priority` property determines the order in which events are delivered to a consumer. This property can be set on all component types; however, it is typically set on individual event messages.

Interaction with OrderPolicy

`Priority` settings are effective only if the delivery points for prioritized messages have their `OrderPolicy` property set to `PriorityOrder` (see "Event Queue Order" on page 72); otherwise, the `Priority` property is ignored. Thus, in order to guarantee that all supplier-assigned priorities are respected in a given channel, `OrderPolicy` must be set to `PriorityOrder` for all proxy suppliers within that channel.

Values

The `Priority` property can be set to any short value between –32,767 (lowest priority) and 32,767 (highest priority), inclusive. By default, all events have a `Priority` setting of 0.

Note: A consumer can modify a message's priority with mapping filters (see "Mapping Filters" on page 93).

Lifetime Properties

Property Names

Lifetime properties specify the time span in which an event remains viable; if the event is not delivered within that time span, it is discarded. By default, events do not have fixed expiry times. The notification service defines three lifetime properties:

- `StopTime`
- `StopTimeSupported`
- `Timeout`

StopTime

`StopTime` sets an absolute expiry time (for example, September 1, 2001), after which the event is no longer deliverable and must be discarded. `StopTime` can only be set in the header of structured event messages.

This property is set with a `TimeBase::UtcT` datatype.

StopTimeSupported

`StopTimeSupported` can be set on a channel, admin, or proxy objects; its boolean setting specifies whether the component supports the `StopTime` property. It has a defualt setting of `TRUE` and the notification service does not currently support a setting of `FALSE`.

Timeout

`Timeout` specifies, in units of 10^{-7} seconds, how long an event remains viable after the channel receives it. After the `Timeout` value expires, the event is no longer deliverable and must be discarded.

You can set this property on a structured event message, channel, admin, or proxy. A consumer can override this property with mapping filters (see "Mapping Filters" on page 93).

This property is set with a `TimeBase::TimeT` datatype; the default value is 0.

Start Time Properties

Property Names

Start time properties specify when an event becomes deliverable. By default, all events are deliverable as soon as they are received by the channel. The notification service defines two start time properties:

- StartTime
- StartTimeSupported

StartTime

StartTime specifies that the event is to be delivered only after the specified time, which is set with a TimeBase::UtcT datatype. This property can only be set on structured event messages.

StartTimeSupported

StartTimeSupported can be set on a channel, admin, or proxy objects, its boolean setting specifies whether the component supports the StartTime property. It has a defualt setting of TRUE and the notification service does not currently support a setting of FALSE.

Undelivered Event Properties

Property Names

Two properties control the behavior of undelivered events in a channel:

- MaxEventsPerConsumer
- DiscardPolicy

MaxEventsPerConsumer

MaxEventsPerConsumer limits the number of undelivered events that a channel queues for a consumer at any given time.

Overflow events are discarded in the order specified by DiscardPolicy.

You can set MaxEventsPerConsumer on:

- Individual consumers, by setting it on their supplier proxies.
- A group of consumers, by setting it on their common consumer admin.
- All consumers connected to a given channel, by setting this property on the channel itself.

This property is set with a long datatype; the default value is 0 (unlimited).

Discard Policy

DiscardPolicy specifies the order in which events are discarded. You can set DiscardPolicy with one of the following constants:

AnyOrder: (default) Discard any events.

FifoOrder: Discard events from the head of the queue.

PriorityOrder: Discard events according to their priority, so lower priority events are discarded before higher priority events.

DeadlineOrder: Discard events in order of shortest expiry deadline first.

LifoOrder: Discard events from the tail of the queue.

Note: Events are discarded only for a consumer whose number of queued events exceeds its MaxEventsPerConsumer setting. The event remains queued for any consumers whose maximum is not exceeded.

RequestTimeout

`RequestTimeout` specifies, in units of 10^{-7} seconds, how much time is allowed a channel object to perform an operation on a client. If the operation does not return within the specified limit, it throws a `CORBA::TRANSIENT` system exception.

This property is set with a `TimeBase::TimeT` datatype; the default is 5 seconds. The maximum value is 600 seconds.

Sequenced Events Properties

Property Names

Consumers that are registered to receive sequences of structured events can control the inflow of events through two properties:

- `MaximumBatchSize`
- `PacingInterval`

Both properties can be set only for supplier proxies of types `SequenceProxyPushSupplier` and `SequenceProxyPullSupplier`. You can set these properties on individual proxies, on consumer admin objects, and on the event channel.

MaximumBatchSize

`MaximumBatchSize` specifies the maximum number of structured events that are sent in a sequence to consumers. This property is set with a `long` datatype; the default value is 1.

PacingInterval

`PacingInterval` specifies, in units of 10^{-7} seconds, the maximum amount of time that a channel is given to assemble structured events into a sequence, before delivering the sequence to consumers. This property is set with a `TimeBase::TimeT` datatype; the default value is 0.

> **Note:** The default values for `MaximumBatchSize` and `PacingInterval` configure a `SequenceProxy` to behave similarly to a `StrucuredProxy`.

Setting Both Properties

With both properties set, a supplier proxy must deliver a sequence of structured events to its consumers when one of the following events occurs:

- The number of events is equal to `MaximumBatchSize`.
- The `PacingInterval` time limit expires.

Proxy Push Supplier Properties

Property Names

Four quality-of-service properties control interaction between a `ProxyPushSupplier` and its consumer:

- `MaxRetries`
- `RetryTimeout`
- `RetryMultiplier`
- `MaxRetryTimeout`

You can set these properties on a `ProxyPushSupplier` on consumer administration objects, and on an event channel.

MaxRetries

`MaxRetries` specifies the maximum number of times that a proxy push supplier calls `push()` on its consumer before it gives up. This property is set with a `CORBA::Ulong` datatype; the default value is 0, which effectivly means an infinite number of retries.

RetryTimeout

`RetryTimeout` specifies, in units of 10^{-7} seconds, how much time elapses between attempts by a proxy push supplier to call `push()` on its consumer. This property is set with a `TimeBase::TimeT` datatype; the default value is 1 second ($1x10^7$).

RetryMultiplier

`RetryMultiplier` specifies the number by which the current value of `RetryTimeout` is multiplied to determine the next `RetryTimout` value. `RetryMultiplier` is applied until either the `push()` is successful or `MaxRetryTimeout` is reached. This property is set with a `CORBA::double` datatype between 1.0 and 2.0; the default value is 1.0.

MaxRetryTimeout

`MaxRetryTimeout` sets the ceiling, in units of 10^{-7} seconds, for `RetryTimeout`. This property applies to `RetryTimeout` values directly assigned by developers as well as `RetryTimeout` values reached by the multiplication of `RetryMultiplier` and `RetryTimeout`. This property is set with a `TimeBase::TimeT` datatype; the default value is 60 seconds ($60x10^7$).

Proxy Pull Consumer Properties

Property Names

Two quality-of-service properties control interaction between a
ProxyPullConsumer and its supplier:

- MaxRetries
- PullInterval

You can set these properties on a ProxyPullConsumer; on supplier admin
objects; and on an event channel.

MaxRetries

MaxRetries specifies the maximum number of times that a proxy pull
consumer calls pull() or try_pull() on its supplier before it gives up. This
property is set with a CORBA::Ulong datatype. The default value is 3.

PullInterval

PullInterval specifies, in units of 10^{-7} seconds, how much time elapses
between attempts by a proxy pull consumer to call pull() or try_pull() on
its supplier. This property is set with a long datatype; the default value is 1
second $(1x10^{7})$.

Channel Administration Properties

Overview

MaxConsumers, MaxSuppliers, MaxQueueLength, and RejectNewEvents apply only to event channel administration, and can be set only on an event channel. These properties are accessible through set_admin() and get_admin().

MaxConsumers

MaxConsumers specifies the maximum number of consumers that can be connected to the channel at any given time. This property is set with a long datatype; the default value is 0 (unlimited).

MaxSuppliers

MaxSuppliers specifies the maximum number of suppliers that can be connected to the channel at any given time. This property is set with a long datatype; the default value is 0 (unlimited).

MaxQueueLength

MaxQueueLength specifies the maximum number of events that will be queued by the channel before the channel begins discarding events or rejecting new events if RejectNewEvents is set to TRUE; the default value is 0 (unlimited).

RejectNewEvents

RejectNewEvents specifies whether or not the channel continues accepting new events after the number of events has reached MaxQueueLength. IONA's implementation only supports a value of TRUE for this property.

When the total number of undelivered events within the channel is equal to MaxQueueLength, each pull-style proxy consumer will stop attempting to perform pull invocations on its supplier until the total number of undelivered events within the channel is decreased. Attempts to push new events to the channel by push-style suppliers will result in the IMPL_LIMIT system exception being raised.

Event Filtering

Filter objects screen events as they pass through the channel, and process those that meet the filter constraints.

Overview

The notification service defines two types of filters:

- *Forwarding filters* are set in a channel by clients that wish to restrict event delivery to those events that meet certain constraints. These filters implement interface `CosNotifyFilter::Filter`.
- *Mapping filters* are set by consumers to adjust the priority or lifetime settings of those messages that meet filter constraints. These filters implement interface `CosNotifyFilter::MappingFilter`.

In This Chapter

The following topics are covered in this chapter:

Forwarding Filters

Overview

Consumers can use forwarding filters to receive only those events that interest them. For example, a consumer within a company's accounting department might use filters to ensure that it receives from government agencies only those events that pertain to tax code changes.

Forwarding filters can be set on individual proxies, both consumer and supplier types, and on groups of proxies through their common admin objects. Because forwarding filters can be set on any delivery point within an event channel, you can build a filtering system that satisfies the individual and collective needs of widely different consumers.

Note: An object that has no filters associated with it forwards all events that it receives to the next delivery point.

In This Section

The following topics are covered in this section:

Implementing a Forwarding Filter

Procedure

Implementing a forwarding filter is a four-step process:

Step	Action
1	Obtain a filter object.
2	Set up filter constraints.
3	Add constraints to the filter object.
4	Attach the filter to a proxy or admin object.

Obtaining a Filter Object

To create filter objects, an application first obtains a filter factory, which is based on interface `CosNotifyFilter::FilterFactory`:

```
// IDL in CosNotifyFilter
interface FilterFactory {
    Filter create_filter (
            in string constraint_grammar)
    raises (InvalidGrammar);
    // ...
};
```

Orbix Notification provides a default filter factory instance that is associated with each event channel. After obtaining a filter factory, the consumer or supplier client calls `create_filter()` on the filter factory object; the call supplies the argument `EXTENDED_TCL`, which specifies the default constraint grammar.

Example

The code in Example 17 obtains a filter object.

Example 17: *Obtaining a Filter Object*

```
// Java
// event channel obtianed earlier
org.omg.CosNotifyFilter.FilterFactory dff =
  channel->default_filter_factory();

org.omg.CosNotifyFilter.Filter filter =
  dff->create_filter("EXTENDED_TCL");
```

Setting Up Filter Constraints

After creating a filter object, you can set up its constraints. Filter objects encapsulate one or more constraints through a sequence of *CosNotifyFilter::ConstraintExp* data structures.

```
// IDL
struct ConstraintExp {
    CosNotification::EventTypeSeq event_types;
    string constraint_expr;
};

typedef sequence<ConstraintExp> ConstraintExpSeq;
```

Each ConstraintExp has two members:

EventTypeSeq specifies a sequence of EventType data structures, each containing two fields that combine to specify an event type:

```
// IDL in module CosNotification
struct EventType {
    string domain_name;
    string domain_type;
};

typedef sequence<EventType>EventTypeSeq;
```

constraint_expr specifies a boolean string expression whose syntax conforms to the default filter constraint language (see "Filter Constraint Language" on page 100).

Example

Example 18 sets up a filter constraint with a single constraint expression, which specifies to forward only even-numbered events:

Example 18: *Setting up a Filter Constraint*

```
// Java
import org.omg.CosNotification.*;
import org.omg.CosNotifyFilter.*;

EventType[] event_types = new EventType(1);

1   event_types[0].domain_name = new String("Orbix Demos");
    event_types[0].type_name =
      new String("Structured Notification Push Demo Event");

2   ConstraintExp[] constraints = ConstraintExp(1);

    constraints[0].event_types = event_types;
3   constraints[0].constraint_expr =
      new String("($EventNumber / 2) == (($EventNumber + 1) / 2)");
```

The filter constraint is set up as follows:

1. A single EventType is initialized, where the domain_name member is set to Orbix Demos; and the type_name member is set to Structured Notification Push Demo Event.

2. A ConstraintExpSeq is defined with a single ConstraintExp member.

3. constraint_expr is set to a boolean string expression, which evaluates to true if an event's $EventNumber is an even integer; false if it is odd.

Adding Constraints to a Filter

After you set up filter constraints, you add them to a filter by calling add_constraints(), as in the following example:

```
org.omg.CosNotifyfilter.ConstraintInfo[] info =
  filter.add_contraints(constraints);
```

The operation checks whether the constraint is syntactically correct; if not, it throws exception InvalidConstraint.

Attaching Filters

All proxy and admin objects inherit `CosNotifyFilter::FilterAdmin`, which provides operations for adding and removing filters:

```
\\ IDL
interface FilterAdmin {
    FilterID add_filter( in Filter new_filter );
    void remove_filter( in FilterID filter )
        raises (FilterNotFound);
    Filter get_filter( in FilterID filter )
        raises (FilterNotFound);
    FilterIDSeq get_all_filters();
    void remove_all_filters();
};
```

You can add one or more filter objects to any proxy or admin object in an event channel, providing multiple filtering layers in a channel.

Example

Example 19 attaches the filter object created earlier to a structured proxy push supplier.

Example 19: *Attach a Filter Object*

```
// Java
import org.omg.CosNotifyChannelAdmin.*;

// event channel ca and filter filter obtained earlier

// create a structured push supplier
ProxySupplier obj =
  ca.obtain_notification_push_supplier
    (ClientType.STRUCTURED_EVENT, proxy_id);
StructuredProxyPushSupplier pps =
  StructuredProxyPushSupplierHelper.narrow(obj);

// add filter to proxy
IntHolder fid = new IntHolder();
fid = pps.add_filter(filter);
```

In this example, the filter is attached to a supplier proxy, so it applies to all events that are targeted at that proxy's consumer. Filters that are attached to an admin object apply to all the admin's proxies. If a set of proxies can use the same filters, it is more efficient to set these on a common admin, so filter processing on a given event takes place only once for all proxies.

If filters are set on an admin and one of its proxies, events can be evaluated against both sets of filters, depending on whether the admin object was created with AND or OR semantics (see "Traversing Multiple Filters in a Channel" on page 90).

Filter Evaluation

A filter evaluates an event against its set of constraints until one evaluates to true. A constraint evaluates to true when both of the following conditions are true:

- A member of the constraint's EventTypeSeq matches the message's event type.
- The constraint expression evaluates to true.

The first filter in which the event message evaluates to true forwards the event to the next delivery point in the channel. If the event message fails to pass any forwarding filters, the event may not be forwarded. For full details on filter processing, see "Processing Events with Forwarding Filters" on page 90.

Processing Events with Forwarding Filters

Overview

When an event message enters an event channel, it can encounter filters at one or more delivery points. The filters at each delivery point evaluate the event message, then either forward the event message to the next delivery point, or drop the event.

Event Message Evaluation

When an object receives an event, it invokes the appropriate match operation—`match_structured()` on structured events, `match()` on untyped events—on its filters. The match operation accepts as input the contents of the event, evaluates it against the filter constraints, and returns a Boolean result:

- `true`: The event satisfies one of the filter constraints and is forwarded immediately to the next delivery point. Other filters for that object are ignored.
- `false`: The event satisfies none of the filter constraints. If the object has multiple filters, the event is passed on to the next filter and the match operation is invoked on it. If all match invocations return false, the event message may be removed from the event channel, depending on the status of its progress in the channel delivery path.

Traversing Multiple Filters in a Channel

Forwarding filters can be attached to admin and proxy objects on both supplier and consumer sides of an event channel. As shows, an event message can potentially traverse four sets of forwarding filters, set on the following objects:

- Consumer proxy
- Supplier admin object
- Consumer's admin object

- Supplier proxy

Event channel
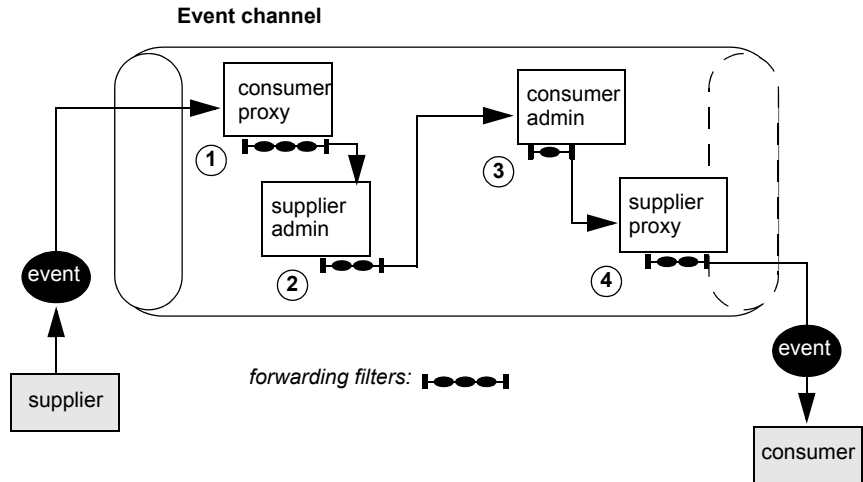
Figure 12: *Forwarding Filters Can Intercept an Event Message at Multiple Delivery Points*

If filters are set on an admin and one of its proxies, events can be evaluated against both sets of filters, depending on whether the admin object was created with AND or OR semantics:

- AND semantics require events to pass both admin and proxy filters.
- OR semantics only require an event to pass an admin or proxy filter.

An event message traverses channel filters as follows:

1. The consumer proxy filters each forwarded event with one of the following results:

    ♦ If the supplier admin has OR semantics, an event that passes any proxy filter is forwarded directly to the consumer admin.

    ♦ If the supplier admin has AND semantics, an event that passes any proxy filter is forwarded to the supplier admin for further filtering.

    ♦ If the admin has AND semantics, an event that fails all proxy filters is not forwarded.

2. The supplier admin filters each event with one of the following results:

- ♦ The event passes one of the filters and is forwarded to the consumer admin.
- ♦ The event fails all filters and is not forwarded.

3. The consumer admin filters each forwarded event with one of the following results:

- ♦ If the admin has OR semantics, an event that passes any filter is forwarded directly to the consumer.
- ♦ If the admin has AND semantics, an event that passes any filter is forwarded to the supplier proxy for further filtering.
- ♦ If the admin has AND semantics, an event that fails all filters is not forwarded.

4. The supplier proxy filters each forwarded event with one of the following results:

- ♦ The event passes one of the filters and is forwarded to the consumer.
- ♦ The event fails all filters and is not forwarded to the consumer.

Mapping Filters

Overview

An event's lifetime and priority can be set at several levels—in the event message itself, and at the channel, admin, or proxy levels. While suppliers can set an event's priority or lifetime—typically, in the header of a structured event message—they cannot always anticipate the importance that individual consumers might assign to events of certain types. For example, a consumer might wish to raise the priority of all messages where `event_type` field is set to `sport` and `sport_type` field is set to `baseball`. Mapping filters allow consumers to increase or diminish the importance of certain events by enabling their supplier proxies to override their `Priority` and `Timeout` properties.

You can apply mapping filters to supplier proxies and consumer admin objects. Each object can have up to two mapping filters:

- A priority filter that determines an event's priority.
- A lifetime filter that determines how long an event remains deliverable.

In This Section

The following topics are discussed in this section:

Implementing a Mapping Filter Object

Procedure

Implementing a mapping filter is a four-step process:

Step	Action
1	Obtain a filter object.
2	Set up constraints and associated values.
3	Add constraints to the filter object.
4	Associate the mapping filter with a supplier proxy or consumer admin.

Obtaining a Mapping Filter Object

To create mapping filter objects, an application first obtains a filter factory, which is based on interface `CosNotifyFilter::FilterFactory`:

```
\\ IDL in module CosNotifyFilter
interface FilterFactory {
    // ...
    MappingFilter create_mapping_filter (
            in string constraint_grammar,
            in any default_value)
        raises(InvalidGrammar);
};
```

The consumer client calls `create_mapping_filter()` on the filter factory object and supplies two arguments:

- The argument `EXTENDED_TCL`, which specifies the default constraint grammar.
- An `any` that specifies the mapping filter's default value. This value is used only when an event message fails to match any filter constraints, and the target property is not set anywhere for the event (see "Processing Events with Mapping Filters" on page 98). This value must be consistent with the mapping filter's target property.

Example

Example 20 creates a mapping filter object and sets its default value to 2.

Example 20: *Creating a Mapping Filter*

```
// Java
// channel obtained earlier
import org.omg.CORBA.*;
import org.omg.CosNotifiyFilter.*;

// channel obtained earlier
FilterFactory_var dff = channel.default_filter_factory();

// set filters default priority to two
Any default_value = ORB.create_any();
default_value.insert_short(2);

//Create filter
MappingFilter Mapfilter =
  dff.create_mapping_filter("EXTENDED_TCL", default_value);
```

Setting Up Filter Constraints

After creating a mapping filter object, you can set up its constraints. Mapping filter objects encapsulate one or more constraints through a sequence of `CosNotifyFilter::MappingConstraintPair` data structures:

```
// IDL in module CosNotifyFilter
// ...
struct ConstraintExp {
    CosNotification::EventTypeSeq event_types;
    string constraint_expr;
};

struct MappingConstraintPair{
    ConstraintExp constraint_expression;
    any result_to_set;
};
```

Each `MappingConstraintPair` contains:

- A constraint that is defined through a `ConstraintExp` data structure (see "Event Type Filtering" on page 102).
- The property override value associated with the constraint. The override value must be consistent with the target property: `short` for a priority filter; `TimeBase::TimeT` for a lifetime filter.

Example

Example 21 sets up a mapping filter constraint with two MappingConstraintPair data structures, which evaluates all events whose event type domain field is set to SportsNews:

- If the event type is set to BaseballResults, and the event's priority is less than 100, reset the priority to 100.
- If the event type is set to FootballResults and the event's priority is greater than 0, reset the priority to 0.

Example 21: *Adding Mapping Filter Constraints*

```java
// Java
import org.omg.CosNotification.*;
import org.omg.CosNotifyFilter.*;


MappingConstraintPair[] mapex = new MappingContsraintPair(2);

// Create first constraint
mapex[0].constriant_expression = new ConstraintExpr(1);
mapex[0].constraint_expression[0].event_types =
  new EventType(1);
mapex[0].contraint_expression[0].event_types[0].domain_name =
  new String("SportsNews");
mapex[0].constraint_expression[0].event_types[0].type_name =
  new String("BaseballResults");
mapex[0].contraint_expression[0].constraint_expr =
  new String("($Priority < 100)");
mapex[0].result_to_set = ORB.create_any();
mapex[0].result_to_set.insert_short(100);

// Create second constraint
mapex[1].constriant_expression = new ConstraintExpr(1);
mapex[1].constraint_expression[0].event_types =
  new EventType(1);
mapex[1].contraint_expression[0].event_types[0].domain_name =
  new String("SportsNews");
mapex[1].constraint_expression[0].event_types[0].type_name =
  new String("FootballResults");
mapex[1].contraint_expression[0].constraint_expr =
  new String("($Priority > 0)");
mapex[1].result_to_set = ORB.create_any();
mapex[1].result_to_set.insert_short(0);
```

Adding Constraints to a Mapping Filter

After you set up filter constraints, you add them to the mapping filter by calling add_mapping_constraints(), as in the following example:

```
org.omg.CosNotifyFilter.MappingConstraintInfo[] mcis1 =
  Mapfilter.add_mapping_constraints(mapexp);
```

The operation checks whether the constraint is syntactically correct; if not, it throws exception InvalidConstraint.

Attaching Mapping Filters

Any supplier proxy and consumer admin can have up to two mapping filters; one that pertains to an event's Priority property, the other to its Timeout property. The following objects provide a method for setting each filter type:

- priority_filter() attaches a mapping filter that can override an event's Priority setting.
- lifetime_filter() attaches a mapping filter that can override an event's Timeout setting.

For example, the following code attaches a priority mapping filter to a supplier proxy:

```
// add the filter to the structured push supplier proxy
pps.priority_filter(Mapfilter);
```

Processing Events with Mapping Filters

Overview

When an event message enters an event channel, it can encounter mapping filters at one or more delivery points. The mapping filters at each delivery point evaluate the event message, and either override the messages quality-of-service settings, set the messages default quality-of-service settings, or do nothing.

Event Evaluation

When a consumer admin or supplier proxy object receives an event, it invokes the appropriate match operation on its mapping filters—`match_structured()` on structured events, `match()` on any-type events:

```
// IDL in interface CosNotifyFilter::MappingFilter
boolean match (in any filterable_data, out any result_to_set)
    raises (UnsupportedFilterableData);

boolean match_structured (
        in CosNotification::StructuredEvent filterable_data,
        out any result_to_set)
    raises (UnsupportedFilterableData);
```

The match operation accepts as input the contents of the event, and evaluates it against the filter constraints. Filter constraints are traversed in descending order of override values—longest-to-shortest lifetime for a lifetime filter, and largest-to-smallest integer for a priority filter.

The match operation returns from each filter with a Boolean result:

- *true*: The event satisfies one of the mapping filter constraints and applies that constraint's override value to the event. The match operation's output parameter returns with the override value.

- *false*: The event satisfies none of the filter constraints. In this case, the event retains its current property setting, if this is explicitly set elsewhere in the channel—for example, by the event channel itself, or in the current proxy. If the target property is not set anywhere, the mapping filter's default value is applied.

While mapping filters effectively change an event's lifetime and priority, they have no effect on event message content. Because they do not depend on finding property settings in the message itself, you can apply mapping filters to any-type and structured event messages alike.

Traversing Multiple Mapping Filters in a Channel

Mapping filters can be attached to a consumer admin and its supplier proxies. If set on both, a supplier proxy's mapping filters take precedence.

Filter Constraint Language

Overview

The default filter constraint language is based on the standard OMG Trader Constraint Language with some modifications that make it more suitable for use as a filter constraint language.

In This Section

The following topics are discussed in this section:

Constraint Expression Data Structure

Constraint Sequence

Filter objects encapsulate one or more constraints through a sequence of `CosNotifyFilter::ConstraintExp` data structures:

```
\\ IDL in module CosNotifyFilter
struct ConstraintExp{
    CosNotification::EventTypeSeq event_types;
    string constraint_expr;
};

typedef sequence<ConstraintExp> ConstraintExpSeq;
```

Each `ConstraintExp` has two members:

- `EventTypeSeq`
- `constraint_expr`

EventTypeSeq

A sequence of `EventType` data structures which contains two fields that specify an event type:

```
\\ IDL
struct EventType {
    string domain_name;
    string domain_type;};

typedef sequence<EventType>EventTypeSeq;
```

constraint_expr

A boolean string expression whose syntax conforms to the default filter constraint language (see "Examples of Notification Service Constraints" on page 108). The constraint expression is applied to events whose event type matches one of the event types defined in the constraint's `EventTypeSeq`.

For full details on the filter constraint language, see the OMG's *Notification Service Specification*.

Event Type Filtering

Overview

The ConstraintExp portion of a constraint is a sequence of EventType data structures identifying which event types are to be filtered. Any event type not specified in a filter's ConstraintExp will be evaluated to false by the filter.

In This Section

The following topics are discussed in this section:

Filtering for a Single Event Type	page 102
Applying a Constraint to All Events	page 102
Using Wildcards	page 103

Filtering for a Single Event Type

Example 22 sets up a constraint expression that evaluates to true for all sports news events reporting on baseball results and whose priority is set to less than 100.

Example 22: *Using the Filter Constraint Language*

```
// Java
import org.CosNotification.*;
import org.omg.CosNotifyFilter.*;

ConstraintExpr[] constriant_expression = new ConstraintExpr(1);
constraint_expression[0].event_types = new EventType(1);
contraint_expression[0].event_types[0].domain_name =
  new String("SportsNews");
constraint_expression[0].event_types[0].type_name =
  new String("BaseballResults");
contraint_expression[0].constraint_expr =
  new String("($Priority < 100)");
```

Applying a Constraint to All Events

A constraint can set its EventTypeSeq to indicate that the constraint expression applies to all events, in several ways:

- Declare an empty EventTypeSeq:

```
org.omg.CosNotification.EventType[] event_types =
  new org.omg.CosNotification.EventType(0);
```

- Initialize a single-element EventTypeSeq to empty strings:

```
org.omg.CosNotification.EventType[] event_types =
  new org.omg.CosNotification.EventType(1);
event_types[0].domain_name = new String("");
event_types[0].type_name = new String("");
```

- Initialize a single-element EventTypeSeq with wildcard characters, *:

```
org.omg.CosNotification.EventType[] event_types =
  new org.omg.CosNotification.EventType(1);
event_types[0].domain_name = new String("*");
event_types[0].type_name = new String("*");
```

Using Wildcards

The default constraint grammar supports wildcard characters in EventType fields. For example, the following setting applies to all news events, such as SportsNews or FinancialNews:

```
org.omg.CosNotification.EventType[] event_types =
  new org.omg.CosNotification.EventType(1);
event_types[0].domain_name = new String("*News");
event_types[0].type_name = new String("*");
```

Referencing Filtered Data

Overview

You can identify any data component in a structured event message by specifying its full path within a CosNotification::StructuredEvent:

```
$.EventHeader[.intermediate-component[...]].component-name
```

For example, you can reference an event type's domain name as follows:

```
$.EventHeader.FixedEventHeader.event_type.domain_name
```

In This Section

The following topics are discussed in this section:

Name-Value Pair Notation

Structured event messages are set up to allow extensive use of name-value pairs sequences. The full syntax for referencing these is as follows:

```
$.EventHeader.FixedEventHeader.OptionalHeaderFields[i].name
$.EventHeader.FixedEventHeader.OptionalHeaderFields[i].value
$.FilterableEventBody[i].name
$.FilterableEventBody[i].value
```

Given this syntax, you can construct a constraint expression that evaluates as follows:

```
($.EventHeader.FixedEventHeader.OptionalHeaderFields[i].name ==
 'Priority') and
($.EventHeader.FixedEventHeader.OptionalHeaderFields[i].value >
 10)
```

While this syntax lets you loop through all optional header and filterable data field members, it is also cumbersome. Therefore, the notification service also supports two abbreviated formats for referencing name-value pairs in a structured event message.

Optional header fields can be represented as follows:

```
$.EventHeader.variable_header(prop-name)
```

For example, the constraint expression shown earlier might be rewritten as follows:

```
$.EventHeader.variable_header(Priority) > 10
```

Filterable data fields can be represented as follows:

```
$.filterable_data(field-name)
```

For example, the following notation refers to filterable data field
StockSymbol:

```
$.filterable_data(StockSymbol)
```

Shorthand Notation

The notification service supports a shorthand notation that lets you reference filterable data components in both structured and unstructured events:

```
$component-name
```

This notation is valid for referencing the following structured event components:

```
$.EventHeader.FixedEventHeader.event_type.domain_name
$.EventHeader.FixedEventHeader.event_type.type_name
$.EventHeader.FixedEventHeader.event_name
$.EventHeader.variable_header.(prop-name)
$.filterable_data.(field-name)
```

For example, the following constraint:

```
($.EventHeader.FixedEventHeader.event_type.type_name ==
 'StockAlert') and
($.EventHeader.variable_header(pct_change) > 5.0)
```

can be rewritten as follows:

```
($type_name == 'StockAlert') and ($pct_change > 5.0)
```

105

The notification service uses the following algorithm to resolve runtime variable $variable:

1. If the variable name is reserved—for example, $curtime—this usage takes precedence.

2. The first matching translation is chosen from:

   - A member of $.EventHeader.FixedEventHeader

   - A property in $.EventHeader.variable_header

   - A field name in $.filterable_data

3. If no match is found, the translation defaults to $.variable.

Thus, a generic constraint can use $Priority to reference an unstructured event's $.priority member, and a structured event's $.EventHeader.variable_header(priority) member.

Operand Handling

Overview

When you add a constraint to a filter, the notification service only checks whether it is syntactically correct. When a filter processes an event, the match operation is responsible for ensuring that operands have valid data types. When the match operation encounters invalid operands, or nonexistent identifiers, it returns false.

Examples

The following constraint expression evaluates three event message fields, a, b, and c:

```
($a + 1 > 32) or ($b == 5) or ($c > 3)
```

The following examples show how the match operation handles constraint operands as it evaluates the contents of different events.

Event 1: <$a, 'Hawaii'>, <$c, 5.0>

The first expression resolves to (Hawaii' + 1 > 32). Because it is not possible to add an integer to a string data type, the constraint is invalid and the match operation returns false.

Event 2: <$a, 5>, <$c, 5.0>

The first expression evaluates to false. Because the event lacks a $b member, an error occurs and the match operation returns false. The constraint expression can be modified to handle the missing $b member as follows:

```
($a + 1 > 32) or (exist $b and $b == 5) or ($c > 3)
```

Event 3: <$a, 5>, <$b, 5.0>

The second expression evaluates to true, although $b is set to a floating point. Following arithmetic conversion rules, the constraint expression's constant 5 is also cast to floating point. Because the second expression evaluates to true, the match operation never detects the omission of member $c.

Examples of Notification Service Constraints

The following examples show different constraint expressions that use the default constraint language:

Accept all `CommunicationsAlarm` events but no `lost_packet` messages:

```
$event_type == 'CommunicationsAlarm' and
not ($event_name == 'lost_packet')
```

Accept `CommunicationsAlarm` events with priorities ranging from 1 to 5:

```
($event_type == 'CommunicationsAlarm') and
($priority >= 1) and ($priority <= 5)
```

Select `MOVIE` events featuring at least three of the Marx Brothers:

```
($event_type == 'MOVIE') and ((('groucho' in $.starlist) +
('chico' in $.starlist) + ('harpo' in $.starlist) +
('zeppo' in $.starlist) + ('gummo' in $.starlist)) > 2)
```

Accept only recent events:

```
$origination_timestamp.high + 2 < $curtime.high
```

Accept students that took all three tests and had an average score of at least 80%:

```
($.test._length == 3) and ((($.test[1].score + $.test[2].score +
                            $.test[3].score) / 3) >= 80)
```

Select processes that exceed a certain usage threshold:

```
$memsize / 5.5 + $cputime * 1275.0 + $filesize * 1.25 > 500000.0h
```

Subscribing and Publishing

Notification service consumers can tell an event channel which event types they wish to receive from suppliers, and suppliers can advertise the event types they offer to consumers.

Overview

The event channel maintains all information about event type supply and demand, and passes this information to consumers and suppliers:

- As consumers change their subscriptions, the channel updates its *subscription list* and informs suppliers of the changes, so that they can adjust event output accordingly.
- As suppliers add or remove event types that they supply, the channel updates its *publication list* and informs consumers of the changes, so that they can re-evaluate their subscriptions.

In This Chapter

The following topics are covered in this chapter:

Event Subscription

Overview

Event subscription enables clients to inform suppliers which events they are interested in receiving. Event subscription requires the following actions from client consumers and suppliers:

- Each consumer subscribes to its desired event types by adding or modifying forwarding filters to their proxy suppliers or consumer admin.
- Each supplier builds its own list of event types to evaluate changes to the channel subscription list against the list of events that they supply.
- On connecting to the event channel, suppliers call `obtain_subscription_types()` on their proxy consumers to discover which event types are currently subscribed to by consumers.
- The supplier's implementation of `subscription_change()` evaluates changes to the channel's subscription list and acts accordingly.

In This Section

The following topics are covered in this section:

Adding Forwarding Filters	page 111
Obtaining Subscriptions	page 113
Implementing subscription_change()	page 116

Adding Forwarding Filters

Overview

A consumer initially specifies which event types it wishes to subscribe to by adding forwarding filters to its proxy supplier or consumer admin. The event types specified in these filters are relayed to the channel, which consolidates, in a single subscription list, all event types that consumers require. A consumer can also remove or modify existing filters. Each time a consumer changes its forwarding filters, the channel modifies its subscription list accordingly.

Filter Modification Operations

A consumer modifies its forwarding filters through one of the following operations, defined in module `CosNotifyFilter`:

in FilterAdmin interface:	in Filter interface:
`add_filter()`	`add_constraints()`
`remove_filter()`	`modify_constraints`
`remove_all_filters();`	`remove_constraints`

Subscription List

The channel's subscription list contains one entry for each event type, and associates a reference count with it. When a consumer adds an unknown event type to one of its filters, the channel opens a new entry in the subscription list and assigns it a reference count of 1. It then notifies client suppliers of the new event type by calling `subscription_change()`, which is implemented by each supplier's developer, on them. The supplier's implementation (see "Implementing subscription_change()" on page 116.) typically uses subscription information to evaluate consumer demand, and to determine whether it should continue or stop supplying certain events.

If an event type's reference count falls to 0—that is, no filters specify this event type—the channel removes the event type from its subscription list. It then notifies all suppliers of the removal through `subscription_change()`. Given this new information, suppliers can stop supplying this particular event type.

Note: Consumers should never invoke `subscription_change()` on their proxy suppliers. The notification service calls this operation automatically when a proxy supplier detects changes in consumer subscriptions.

For information about implementing forwarding filters, see "Forwarding Filters" on page 84.

Example

Example 23 implements a client push supplier that defines an array of SuppliedType elements. This structure encapsulates the event types that this supplier can produce, and sets a flag of true or false to indicate which ones the supplier should push.

Example 23: *Client Push Supplier*

```Java
// Java
class NotifyPushSupplier extends StructuredPushSupplierPOA
{
  class SuppliedType
    {
       public String domain_name;
       public String type_name;
       public boolean supply;
    }

private SuppliedType[] m_supply_types = null;
private int num_types_supplied = 5;
//...
}
```

Obtaining Subscriptions

obtain_subscription_types()

After a supplier connects to an event channel, it can ascertain which event types consumers currently require by calling `obtain_subscription_types()` on its proxy consumer. This operation is defined as follows:

```
org.omg.CosNotification.EventType[]
  obtain_subscription_types(in ObtainInfoMode mode);
```

Arguments

It takes a single `ObtainInfoMode` argument as input, which informs the channel whether to automatically notify this supplier of future subscription list changes. This argument is typically set with one of the following flags:

ALL_NOW_UPDATES_ON: The invocation returns the contents of the subscription list, and enables automatic notification by `subscription_change()`. Use this argument for a supplier that implements `subscription_change()` to handle notification (see "Implementing subscription_change()" on page 116).

ALL_NOW_UPDATES_OFF: The invocation returns the contents of the subscription list, and disables automatic notification. Use this argument for a supplier that wishes to control when it receives subscription changes, through subsequent calls to `obtain_subscription_types()`.

NONE_NOW_UPDATES_ON: The invocation enables automatic notification of updates to the subscription list without returning the contents of the subscription list. Use this argument for a supplier that implements `subscription_change()` to handle notification (see "Implementing subscription_change()" on page 116).

NONE_NOW_UPDATES_OFF: The invocation disables automatic notification of updates to the subscription list without returning the contents of the subscription list. Use this argument for a supplier that wishes to control when it receives subscription changes, through subsequent calls to `obtain_subscription_types()`.

Return Values

The operation returns an `EventType[]`, which contains all event types currently requested by consumers.

Example

In Example 24, a client supplier performs the following steps to implement `obtain_subscription_types()`:

1. Initializes a list of event types that it supplies.

2. Calls `obtain_subscription_types()` to obtain a list of subscription types.

3. For each subscription type, calls `find_index()` (shown in the next section), which compares each subscription type against the client's own event types list; if the event types match, it sets the list element's `supply` flag to true.

Example 24: Implementing `obtain_subscription_types()`

```java
// Java
class NotifyPushSupplier extends StructuredPushSupplierPOA
{
\\ ...

void init_supply_types()
  {
    int i;
    m_supply_types = new SupplyType[num_types_supplied];

    for(i=0; i < num_types_supplied ; i++)
      {
        m_supply_types[i].domain_name =
          new String("SportsNews");
        m_supply_types[i].supply = false;
      }

    m_supply_types[0].type_name = new String("BaseBallResults");
    m_supply_types[1].type_name = new String("FootballResults");
// other sporting events ...
    m_supply_types[4].type_name = new String("TennisResults");

    org.omg.CosNotification.EventType[] types_to_supply =
      m_proxy->obtain_subscription_types(ALL_NOW_UPDATES_ON);
```

1 (marks line: `m_supply_types = new SupplyType[num_types_supplied];`)

2 (marks line: `org.omg.CosNotification.EventType[] types_to_supply =`)

Example 24: Implementing `obtain_subscription_types()`

```
3        // For each supplied event type which consumers want,
         // set its boolean flag to true
         long index;

         for (i=0; i<types_to_supply.length(); i++)
         {
             index = find_index(types_to_supply[i]);
             if (index >= 0)
                 m_supply_types[index].supply = true;
         }
     }

     // ...
     }
```

Implementing subscription_change()

subscription_change()

When the channel's subscription list adds or removes an event type, the channel automatically calls `subscription_change()` on all client suppliers. This operation is defined in interface `CosNotifyComm::NotifySubscribe`:

```
\\ IDL
module CosNotifyComm
{
  exception InvalidEventType {CosNotification::EventType
  type;};
  // ...
  interface NotifySubscribe
  {
    void subscription_change(
         in CosNotification::EventTypeSeq added,
         in CosNotification::EventTypeSeq removed)
      raises (InvalidEventType);
  }
  // ...
};
```

Arguments

The operation receives two `EventTypeSeq` arguments:

added   Specifies new event types that this supplier now offers.

removed   Specifies event types that the supplier no longer offers.

A supplier implements this operation in order to ascertain which event types are being consumed and which are not, and reevaluate its event output accordingly.

Example

If a consumer subscribes to sports news events, suppliers can detect this interest through their implementation of `subscription_change()` and start to push events of that type. When consumers are no longer interested in this event type, the channel's subscription list changes again, and the channel calls `subscription_change()` on its suppliers with this change. The supplier can then stop pushing those events.

In the implementation of subscription_change() shown in Example 25 the supplier updates the list of events that it can supply.

Example 25: *Updating the Supplier's List of Events*

```java
// Java
class NotifyPushSupplier extends StructuredPushSupplierPOA
{
\\ ...
int find_index(org.omg.CosNotification.EventType e_type)
  {
     for(int i=0; i < num_types_supplied; i++)
       if(etype.domain_name.equals(m_supply_type[i].domain_name)
          && etype.type_name.equals(m_supply_tpye[i].type_name))
        return i;

     return -1;  // event tyoe not found
  }

void subscription_change(org.omg.CosNotifaction.EventType[]
    added, org.omg.EventType[] removed)
  {
     // Turn on supplying of added types
     for(i = 0; i < added.length(); i++)
       if find_index(added[i]) >= 0)
         m_supply_types[i].supply = IT_TRUE;

     // Turn off supplying of removed types
     for(i = 0; i < removed.length(); i++)
       if(find_index(removed[i]) >= 0)
         m_supply_types[i].supply = IT_FALSE;
  }
// ...
}
```

1 (marks the `// Turn on supplying of added types` block)

2 (marks the `// Turn off supplying of removed types` block)

This code executes as follows:

1. The first argument (added) is evaluated for new event types that have been added to the subscription list. If the argument contains event types, find_index() is called for each event type and compares it against the client's list of supplied event types. If it is on the list, the event type's Boolean flag is set to true.

2. The second argument (`removed`) is evaluated for event types that have been removed from the subscription list. If the argument contains event types, `find_index()` is called for each event type and compares it against the client's list of supplied event types. If it is on the list, the event type's Boolean flag is set to false.

Note: A supplier that wishes not to be notified of subscription changes should implement `subscription_change()` to throw a `CORBA::NO_IMPLEMENT` exception.

Publishing Event Types

Overview

Event publication enables consumers to discover new event types as they are offered by suppliers. Event publication requires the following actions from client consumers and suppliers:

- Suppliers advertise event types that they can provide by calling `offer_change()`.
- On connecting the consumer to the event channel, consumers call `obtain_offered_types()` on their proxy suppliers to discover which event types are currently available.
- The consumer's implementation of `offer_change()` evaluates changes to the channel's publication list and acts accordingly.

In This Section

The following topics are covered in this section:

Advertising Event Types

offer_change()

A supplier informs the event channel of those event types that it can supply by calling `offer_change()` on its proxy consumer or supplier admin object. This operation is defined in interface `NotifyPublish` interface, which is inherited by all `ConsumerAdmin` and `SupplierAdmin` interfaces:

```
\\ IDL
module CosNotifyComm
{
    exception InvalidEventType{CosNotification::EventType type;};

    interface NotifyPublish
      {
        void offer_change(in CosNotification::EventTypeSeq added,
                          in CosNotification::EventTypeSeq removed)
        raises(InvalidEventType);
    };
    // ...
};
```

Arguments to offer_change()

`offer_change()` receives two arguments of the `EventTypeSeq` type, which is a sequence of `EventType` structures defined as follows:

```
// IDL in module CosNotification
struct EventType {
    string domain_name;
    string type_name;
};

typedef sequence<EventType> EventTypeSeq;
```

The two parameters let the supplier modify the channel's publication list:

added	Specifies new event types that this supplier now offers.
removed	Specifies event types that the supplier no longer offers.

Publication List

An event channel maintains a single publication list of all event types that its suppliers advertise, which it updates with each supplier's invocation of `offer_change()`.

The channel's publication list contains one entry for each event type, and associates a reference count with it. When a supplier calls offer_change() with an unknown event type, the channel opens an entry in the publication list and assigns it a reference count of 1. It then notifies client consumers of the new event type by calling offer_change() on them. The consumer's implementation (see "Implementing offer_change()" on page 126) typically evaluates the updated publication data, to determine whether it contains event types of interest.

As other suppliers advertise the same event type, the channel updates its reference count. However, intermediate changes in an event type's reference count—for example, an increase from 1 to 2—are not conveyed to consumers.

If an event type's reference count falls to 0—that is, no suppliers offer this event type—the channel removes the event type from its publication list. It then notifies all consumers of the removal through offer_change(). Given this new information, consumers can remove or modify the filters that forward this event type, and avoid the overhead these otherwise incur.

Example

In the following code, a supplier builds event types that it wishes to supply, and adds them to an EventType sequence. It then invokes offer_change() on its structured proxy push consumer, structured_ppc.

```Java
// Java
import org.omg.CosNotification.*;

EventType[] added = new EventType[2];
added[0].domain_name = new String("SportsNews");
added[0].type_name = new String("BaseballResults");
added[1].domain_name = new String("SportsNews");
added[1].type_name = new String("FootballResults");

EventType[] removed = EventType[0];

structured_ppc.offer_change(added, removed);
```

Discovering Available Event Types

obtain_offered_types()

After a consumer connects to an event channel, it can ascertain which event types are currently available from suppliers by calling `obtain_offered_types()` on its proxy supplier or consumer admin. This operation is defined as follows:

```
CosNotification::EventTypeSeq
  obtain_offered_types(in ObtainInfoMode mode);
```

Arguments

It takes a single `ObtainInfoMode` argument as input, which informs the channel whether or not to automatically notify this consumer of future publication list changes. This argument is typically set with one of the following flags:

ALL_NOW_UPDATES_ON: The invocation returns the contents of the publication list, and enables automatic notification of future changes to the list through `offer_change()`. Use this argument for a consumer that implements `offer_change()` to handle notification (see "Implementing offer_change()" on page 126).

ALL_NOW_UPDATES_OFF: The invocation returns the contents of the publication list, and disables automatic notification. Use this argument for a consumer that wishes to control when it receives publication changes through subsequent calls to `obtain_offered_types()`.

NONE_NOW_UPDATES_ON: The invocation enables automatic notification of updates to the publication list without returning the contents of the publication list. Use this argument for a supplier that implements `offer_change()` to handle notification (see "Implementing offer_change()" on page 126).

NONE_NOW_UPDATES_OFF: The invocation disables automatic notification of updates to the publication list without returning the contents of the publication list. Use this argument for a supplier that wishes to control when it receives publication changes, through subsequent calls to `obtain_offered_types()`.

Return Values

The operation returns an EventTypeSeq, which contains all event types currently available from suppliers.

Example

The code shown in Example 26 might be called by a consumer during or immediately after instantiation. In it, two methods are implemented.

init_consume_types() calls obtain_offered_types(), which returns with all currently advertised event types. The method then calls get_choices(), which returns with the events selected (if any) by an end user. The method finally calls add_subscription().

add_subscription() receives the user-selected event types and builds a forwarding filter for each one. It then builds an indexed list of filter data and their IDs, which allows the client consumer to access filters as its subscription needs change.

Example 26: *Subscribing to Selected Event Types*

```java
// Java
import org.omg.CosNotifyFilter.*;
import org.omg.CosNotification.*;

class NotifyPushConsumer extends StructuredPushConsumerPOA
{
FilterID[] filterID;
int num_filters;
EventType[] consume_types;
int max_filters = 10;  // hard coded array size

\\ ...
```

Example 26: *Subscribing to Selected Event Types*

```
// Add a subscription for new event types chosen by user
public void add_subscription(EventType e_type)
{
  // Create a filter for the new subscription
  FilterFactory dff = channel.default_filter_factory();
  Filter filter = dff.create_filter("EXTENDED_TCL");

  // Set up constraint expression for new filter
  EventType[] event_types = new EventType(1);
  event_types[0].domain_name = new String(e_type.domain_name);
  event_types[0].type_name = new String(e_type.type_name);

  ConstraintExp[] constraints = new CosNotifyFilter(1);
  constraints[0].event_types = event_types;
  constraints[0].constraint_expr = new String("");

  // Add constraint to new filter
  ConstraintInfo[] info = filter->add_constraints(constraints)

  filterID[num_filters] = proxy.add_filter(filter);

// Update internal data structures to track subscription data
  consume_types[num_filters].domain_name =
    new String(e_type.domain_name);
  consume_types[num_filters].type_name =
    new String(e_type.type_name);
  num_filters++;
}
```

1
```
public void init_consume_types()
{
  org.omg.CosNotification.EventType[] types_available =
    proxy.obtain_offered_types(ALL_NOW_UPDATES_ON);
```

2
```
  // return with user choices
  org.omg.CosNotification.EventType[] types_wanted =
    get_choices(types_available);
```

3
```
  for (int i = 0; i < types_wanted.length(); i++)
    add_subscription(types_wanted[i]);
}

// ...
}// NotifyPushConsumer
```

The code executes as follows:

1. Obtains all available event types that are currently advertised in the event channel.

2. Calls `get_choices()`, which returns with user-selected event types.

3. For each chosen event type, calls `add_subscription()`, which subscribes the client consumer to receive that event type.

Implementing offer_change()

Overview

When the channel's publication list adds or removes an event type, the channel calls `offer_change()` on all client consumers. This operation receives two input arguments of type `EventTypeSeq`, which contain added and removed event types (see "Arguments to offer_change()" on page 120). A consumer's implementation should examine both arguments and re-evaluate its subscriptions accordingly.

Example

In Example 27, `offer_change()` returns new event types to an end user, who decides which (if any) of the new event types to subscribe to.

Example 27: *Adding and Removing Event Types*

```Java
// Java
import org.omg.CosNotification.*;
import org.omg.CosNotifyFilter.*;

class NotifyPushConsumer extends StructuredPushConsumerPOA
{
FilterID[] filterID;
int num_filters;
EventType[] consume_types;
int max_filters = 10;  // hard coded array size

\\ ...

public void offer_change(EventType[] added, EventType[] removed)
throw (org.omg.CORBA.SystemException)
{
  // return with user choices
  EventType[] types_wanted;

  if (added.length() > 0)
    {
      types_wanted = get_choices(added);// not implemented here

      for(int i=0; i < types_wanted.length(); i++)
        add_subscription(types_wanted[i]);
    } // if added
```

Example 27: *Adding and Removing Event Types*

```
2    // Remove subscription for types no longer supplied
     for(int i = 0; i < removed.length(); i++)
       {

       for(int n = 0, n < num_filters; n++)
         {
           if
       (removed[i].domain_name.equals(consume_types[n].domain_name)
        && removed[i].type_name.equals(consume_types[n].type_name))
           {
             // Remove filter from proxy
             proxy.remove_filter(filterID[n]);

             // Remove subscription data from customer list
             for (int ix = n; ix < (num_filters-1); ix++)
               {
                 filterID[ix] = filterID[ix + 1];
                 consume_types[ix].domain_name =
                   consume_types[ix + 1].domain_name;
                 consume_types[ix].type_name =
                   consume_types[ix + 1].type_name;
               } // for ix

               // Resize data structures appropriately.
               num_filters--;

           }// if equals
         } // for n
       } // for i

} // offer_change

// ...
}
```

This code executes as follows:

1. The first argument (`added`) is evaluated for new event types that have been added to the publication list. If the argument contains event types, `get_choices()` is called and returns with the user's choices, if any. For each event type chosen, `add_subscription()` is called (shown in the previous section), which builds a filter for that event type, and updates the consumer's own subscription list.

2. The second argument (`removed`) is evaluated for event types that have been removed from the subscription list. If the argument contains event types, the method looks up each event type in the consumer's subscription list. If found, the corresponding filter is removed and the consumer's subscription list is updated.

> **Note:** A consumer that wishes not to be notified of publication changes should implement `offer_change()` to throw exception `CORBA::NO_IMPLEMENT`.

Multicast Consumers

A group of consumers that subscribe to the same events can connect to the notification service by using a UDP/IP Multicast based protocol, thereby reducing network overhead.

Overview

A notification service with many clients will generate a large amount of network traffic. The Orbix notification service provides a multicast based protocol to reduce the network overhead.

In this Chapter

This chapter covers the following topics:

MIOP

Definition
Multicast Inter-ORB Protocol *(MIOP)* provides one-way communication between the notification service and groups of similar event consumers, using the UDP IP/Multicast protocol. This protocol helps lower network overhead when a large number of push-style consumers are receiving the same events.

Endpoint Groups
With MIOP, any number of push-style consumers interested in receiving identical events can join an *endpoint group*. While TCP/IP based IIOP requires the service to send one message per individual client, IP/Multicast based MIOP only requires one message per endpoint group. The endpoint group members attach to the same proxy supplier, and share the same filters and quality-of-service properties.

Limitations
MIOP cannot verify receipt of events by individual consumers. This raises the possibility that interested consumers using MIOP may miss events due to being unreachable when the channel sends them.

Note: The OMG provides no specifications for MIOP. Therefore, notification services from other vendors might be incompatible with Orbix IP/Multicast consumers.

IDL Interfaces

Interfaces for Endpoint Groups

The module `IT_NotifyComm` extends `CosNotifyComm` and provides interfaces for IP/Multicast endpoint groups. These interfaces support push-style delivery of untyped, structured, and sequence events to endpoint groups, via a UDP IP/Multicast based protocol.

The interfaces that support endpoint groups are defined as follows:

```
// IDL
module IT_NotifyComm
{
  interface GroupNotifyPublish
  {
    oneway void offer_change(
                in CosNotification::EventTypeSeq added,
                in CosNotification::EventTypeSeq removed);
  }; // GroupNotifyPublish

  interface GroupPushConsumer : GroupNotifyPublish
  {
    oneway void push(in any data);
    oneway void disconnect_push_consumer();
  }; // GroupPushConsumer

  interface GroupStructuredPushConsumer : GroupNotifyPublish
  {
    oneway void push_structured_event(
                in CosNotification::StructuredEvent notification);
    oneway void disconnect_structured_push_consumer();
  }; // GroupStructuredPushConsumer

  interface GroupSequencePushConsumer : GroupNotifyPublish
  {
    oneway void push_structured_events(
                in CosNotification::EventBatch notifications);
    oneway void disconnect_sequence_push_consumer();
  }; // GroupSequencePushConsumer
};  // IT_NotifyComm
```

Oneway Communication

The interfaces for multicast consumers only support oneway invocation. MIOP only provides communication from the notification channel to the consumers. Consumers cannot report back to the notification service regarding the success or failure of a given transmission.

Consumers communicate with the notification service via standard IIOP.

Configuring Orbix for Multicast

Configuration Scope

In order to use MIOP, the runtime ORB must load the `egmiop` plug-in. A named configuration scope must be created that establishes the proper settings.

Settings

In order to configure the ORB to load the correct plug-ins for multicast, follow these steps:

1. Include `"egmiop"` in the `orb_plugins` list.
2. Include `"GIOP+EGMIOP"` in the `binding:client_binding_list`.
3. Label the well known addressing id and set `<label>:egmiop:addr_list` property to a valid multicast endpoint address.

When each multicast client starts up, it finds the proper configuration scope by initializing the ORB with a name that corresponds to a multicast configuration scope. Each client must also set its well-known addressing ID to the correct label.

Example

The following configuration excerpt creates a configuration scope for the ORB `egmiop_test`. It includes the plug-in and the bindings required to use multicast. It labels the well-known address `"miop_test"`.

```
egmiop_test
{
  orb_plugins = ["iiop_profile", "giop", "iiop", "egmiop"];
  binding:client_binding_list = ["GIOP+EGMIOP", "POA_Coloc",
                                 "OTS+TLS_Coloc+POA_Coloc",
                                 "TLS_Coloc+POA_Coloc",
                                 "OTS+GIOP+IIOP", "GIOP+IIOP"];
  miop_test:egmiop:addr_list = ["228.0.0.0:500"];
}
```

Implementing an Endpoint Group

Overview

To use MIOP effectively, create an endpoint group of push-style consumers who share identical event subscriptions and quality-of-service properties.

In this Section

This section covers the following topics:

Instantiating an IP/Multicast Consumer

Determining the Interface   Consumers that use IP/Multicast are instantiated from the `IT_NotifyComm` group interface that corresponds to the type of events the group will receive—`any`, `structured`, or `sequence` (see "Interfaces for Endpoint Groups" on page 131).

ORB Initialization   The consumer must also initialize an ORB whose configuration scope establishes the correct environment for MIOP (see "Configuring Orbix for Multicast" on page 133).

Example   Example 28 shows how a client application might instantiate a consumer of type `GroupPushConsumer` and initialize an ORB whose configuration scope loads the correct plug-ins for MIOP.

Example 28: *Instantiating a Consumer for Multicast (Sheet 1 of 2)*

```Java
// Java
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
import org.omg.CosNotification.*;
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.CosNotifyComm.*;
import org.omg.TimeBase.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POAPackage.*;

// Iona imports
import com.iona.corba.IT_Notifucation.*;
import com.iona.corba.IT_NotifyChannelAdmin.*;
import com.iona.corba.IT_NotifyComm.*;

class NotifyPushConsumer extends GroupPushConsumerPOA
{
public static ORB orb;
// member variables not shown...
```

Example 28: *Instantiating a Consumer for Multicast (Sheet 2 of 2)*

```
// The main entry point @param args command line args
  public static void main (String args[])
    {
// Add -ORBname to end of argument list to ensure the proper
  configuration scope
    String[] orb_name_args = new String[args.length + 2];
    System.arraycopy(args, 0, orb_name_args, 0, args.length);
    orb_name_args[orb_name_args.length - 2] = "-ORBname";
    orb_name_args[orb_name_args.length - 1] = "egmiop_test";

    orb = ORB.init(orb_name_args, null);

    //POA initialization not shown ...

    NotifyPushConsumer consumer = new NotifyPushConsumer();

    // ...
   }

  void public NotifyPushConsumer()
    {
    }

  void public ~NotifyPushConsumer()
    {
    }

// ...
}
```

Required Methods

You must provide implementations for `push()`, `offer_change()`, and `disconnect_push_consumer()` for consumers. `IT_NotifyComm` also specifies the methods `disconnect_structured_push_consumer()` and `disconnect_sequence_push_consumer()` for clients that support those event types.

Creating a POA for an Endpoint Group

Required Policies

To create an endpoint group, all of the endpoint group members must create POAs with the following policies:

POA Policy	Setting
PERSISTENCE_MODE_POLICY_ID	DIRECT_PERSISTENCE
LIFESPAN_POLICY	PERSISTENT
ID_ASSIGNMENT_POLICY	USER_ID
WELL_KNOWN_ADDRESSING_POLICY_ID	An agreed upon label as specified in the configuration scope for the ORB (see "Configuring Orbix for Multicast" on page 133).

In addition, every endpoint group member must also use an agreed upon POA name.

Note: If a consumer's POA name is not identical to the POA names of the endpoint group members, it will not become a member of the endpoint group.

Example

The code in Example 29 creates a POA with the correct policies. It must be run by every consumer wishing to join the endpoint group.

Example 29: *Creating a POA for an Endpoint Group (Sheet 1 of 2)*

```
// Java
import org.omg.CORBA*.;
import org.omg.PortableServer*.;

// ...

Object obj = orb.resolve_initial_references("RootPOA");
POA root_poa = POAHelper.narrow(obj);
POAManager poa_manager = root_poa.the_POAManager();
```

Example 29: *Creating a POA for an Endpoint Group (Sheet 2 of 2)*

```
Policy[] policies = new Policy[4];
Any addressing_id = orb.create_any();
addressing_id.insert_string("miop");
policies[0] =
  orb.create_policy(WELL_KNOWN_ADDRESSING_POLICY_ID.value,
                    addressing_id);

policies[1] =
root_poa.create_lifespan_policy(LifespanPolicyValue.PERSISTENT);

Any persistent_mode = orb.create_any();
PersistenceModePolicyValueHelper.insert(persistent_mode,
  PersistenceModePolicyValue.DIRECT_PESISTENCE);
policies[2] =
  orb.create_policy(PERSISTENCE_MODE_POLICY_ID.Value,
                    persistent_mode);

policies[3] =
root_poa.create_id_assignment_policy(IdAssignmentPolicyValue.USE
    R_ID);

POA multicast_poa =
  root_poa.create_POA("miop_poa", poa_manager, policies);
```

Registering an Endpoint Group Object Reference

Object Name

After each endpoint group member creates a POA with the correct policies and name, it must register an object reference. Each endpoint group member registers with the same object reference. All endpoint group members must use the same object name to generate an object reference. Because this group object reference is created with a POA configured to support MIOP, it contains the multicast information needed to reach the endpoint group members.

Note: The consumer's object name must be identical to the other endpoint group member's object names. Otherwise, it will not join the endpoint group.

Example

Example 30 shows how an endpoint group member might register with a group object reference.

Example 30: *Registering with a Group Object Reference*

```
// Java
import org.omg.PortableServer.*;

// ...
1  ObjectId oid = multicast_poa.string_to_ObectId("miopConsumer");
2  multicast_poa.activate_object_with_id(oid, consumer);

   POAManager multicast_poa_manager =
     multicast_poa.the_POAManager();
3  poa_manager.activate();
   multicast_poa_manager.activate();
```

The code executes as follows:

1. Gets an object ID for the consumer using the name agreed on by all members of the group.

2. Registers the consumer's object reference by activating it.

3. Activates the multicast POA to receive messages.

Repeat this sequence for each endpoint group member.

Connecting to an Event Channel

Overview

All endpoint group members share the same proxy supplier. Therefore, only one endpoint group member connects to the channel. After this endpoint group member connects, the group can begin receiving messages.

Because all of the consumers in an endpoint group share a proxy, they also share the same event subscriptions, filters, and quality-of-service properties.

Interfaces

Module `IT_NotifyChannelAdmin` provides an interface to connect endpoint groups of each consumer type—`any`, `structured`, or `sequence`—to a notification channel:

```
// IDL
interface GroupProxyPushSupplier :
    CosNotifyChannelAdmin::ProxyPushSupplier
{
    void connect_group_any_push_consumer(
        in IT_NotifyComm::GroupPushConsumer group_push_consumer)
    raises(
            CosEventChannelAdmin::AlreadyConnected,
            CosEventChannelAdmin::TypeError
            );
};  // GroupProxyPushSupplier

interface GroupStructuredProxyPushSupplier :
        CosNotifyChannelAdmin::StructuredProxyPushSupplier
{
  void connect_group_structured_push_consumer(
      in IT_NotifyComm::GroupStructuredPushConsumer
        group_push_consumer)
  raises(
        CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError
        );
};  // GroupStructuredProxyPushSupplier
```

```
interface GroupSequenceProxyPushSupplier :
        CosNotifyChannelAdmin::SequenceProxyPushSupplier
{
  void connect_group_sequence_push_consumer(
       in IT_NotifyComm::GroupSequencePushConsumer
          group_push_consumer)
  raises(
        CosEventChannelAdmin::AlreadyConnected,
        CosEventChannelAdmin::TypeError
        );
};  // GroupSequenceProxyPushSupplier
```

Implementation

The connecting consumer creates a group proxy supplier of the same type in a notification channel. It then connects to the event channels by invoking the corresponding connect operation on the proxy.

> **Note:** If more than one member of the group attempts to connect to the event channel, an AlreadyConnected exception is raised.

Group Proxy

The proxy created by the connecting consumer serves as the proxy for the entire endpoint group and is shared by all of the endpoint group members.

If the connecting consumer disconnects from the channel, all members of the endpoint group also disconnect. However, if the connecting consumer dies without disconnecting, the proxy remains active and the remaining members of the group continue to receive events.

Example

Example 31 shows how to connect an endpoint group of
GroupPushConsumers to a notification channel.

Example 31: *Connecting an Endpoint Group to an Event Channel (Sheet 1
of 2)*

```
// Java
import org.omg.CORBA.*;
import org.omg.CosEventChannelAdmin.*;
import org.omg.CosNotification.*;
import org.omg.CosNotifyChannelAdmin.*;

//IONA specific classes
import com.iona.corba.IT_NotifyComm.*;
import com.iona.corba.IT_NotifyChannelAdmin.*;

class NotifyPushConsumer extends GroupPushConsumerPOA
{
public static ORB orb = null;
public static EventChannel ec = null;
// member variables not shown...
// ...

void main(String[] args)
{
  // ORB and POA activation shown above
  // ...

  org.omg.CORBA.Object obj =
   orb.resolve_initial_references("NotificationService");
  EventChannelFactory factory =
    EventChannelFactoryHelper.narrow(obj);

  IntHolder id = new IntHolder();
  Property[] init_qos = new Property[0];
  Property[] init_admin = new Property[0];
```

1

Example 31: *Connecting an Endpoint Group to an Event Channel (Sheet 2 of 2)*

```
2    try
       {
         ec = factory.create_named_channel("miop_channel", init_qos,
                                           init_admin, id);
       }
     catch(ChannelAlreadyExists cae)
       {
       // channel already exists
         try
           {
             ec = factory.find_channel("miop_channel", id);
           }
         catch(ChannelNotFound cnf)
           {
             System.err.println("Could not create or find
                                 notification channel.");
             System.exit(1);
           } // catch(ChannelNotFound)
       } // catch(ChannelAlreadyExists)

3    InterFilterGroupOperator op = new InterFilterGroupOperator();
     op = CosNotifyChannelAdmin.AND_OP;
     AdminID id;
     ConsumerAdmin ca = ec.new_for_consumers(op, id);

     IntHolder proxy_id = new IntHolder();
     ClientType ctype = ClientType.ANY_EVENT;

4    ProxySupplier obj =
       ca.obtain_notification_push_supplier(ctype, proxy_id);
     GroupProxyPushSupplier pps =
       GroupProxyPushSupplierHelper.narrow(obj);

5    try
       {
         pps.connect_group_any_push_consumer(consumer);
       }
     catch(AlreadyConnected)
       {
         // implementation left to developer
       }
   } // main
   } // NotifyPushConsumer
```

This code executes as follows:

1. Obtains an `EventChannelFactory` from the ORB.
2. Obtains the event channel `miop_channel`.
3. Creates a consumer admin object for the group.
4. Creates a `GroupProxyPushSupplier` for the group.
5. Invokes `connect` on the consumer and catches any exceptions.

Receiving Events

Consumers that use IP/Multicast receive events the same way as a non-multicast, push-style consumer (see "Obtaining Event Messages" on page 52).

Filtering and Event Subscription

Overview

Like non-multicast consumers, endpoint groups can use mapping and forwarding filters and subscribe to events. However, because they share a proxy supplier, any change in filters or subscriptions made by one endpoint group member affects every other endpoint group member.

ALL_UPDATES_NOW

To maximize the overhead benefits of using IP/Multicast, consumers should call `obtain_offered_types()` with `ALL_UPDATES_NOW`. The channel then automatically notifies the group of future changes in the list of available events using IP/Multicast through `offer_change()`. Consumers should implement `offer_change()` to handle notification (see "Implementing offer_change()" on page 126).

Updating the Subscription List

Changes to the list of available events are broadcast to all endpoint group members using this implementation. However, only one endpoint group member should make changes to the subscription list, because all endpoint group members share the same proxy.

IIOP Calls

Calls to `obtain_offered_events()`, `create_filter()`, and `add_filter()` are two-way and do not use IP/Multicast.

For More Information

For more information on filters and subscribing to events, see "Event Filtering" on page 83 and "Subscribing and Publishing" on page 109.

Disconnecting from an Event Channel

Overview

An endpoint group is disconnected from the notification channel when one of its members invokes the disconnect operation on the group's proxy supplier. This operation terminates the connection between the group and its proxy supplier. The notification channel then releases all resources allocated to support its connection to the group, including the destruction of the group's proxy supplier.

Each proxy supplier interface supports a disconnect operation. For example, `disconnect_structured_push_supplier()` is defined in `StructuredProxyPushSupplier`.

> **WARNING:** When one group member invokes disconnect on the proxy supplier, all members of the group stop receiving events from the notification channel.

Managing the Notification Service

Orbix notification provides several configuration variables that allow you to control the behavior of a deployed notification service.

In this chapter

This chapter discusses the following topics:

Configuring the Notification Service

Uses of Configuration Variables

Configuration variables allow the user to control the behavior of the notification service. You can alter the number of event channels that can be created, the maximum number of notification clients, the threading behavior of the individual components of the service, and other properties. Because the elements in the notification service are interdependent, changing one configuration variable may affect how several components of the service perform.

Namespaces

The notification service's behavior is affected by variables in two namespaces:

plugins:notification The variables in this namespace control both the event and notification service. They control the general performance characteristics of event channel objects, including the number of threads they can use and how many event channels can be created at a time.

plugins:notify The variables in this namespace are specific to the notification service. They control the amount of debugging information the notification service generates, how the service's database behaves, and the threading strategy used in dispatching events to notification service clients.

For a complete listing of the notification service's configuration variables, see the *CORBA Administrator's Guide*.

Changing

You can edit the values of the notification service's configuration variables either by using `itadmin` or, in the case of a file-based configuration, hand editing the configuration file. For more information, see the *CORBA Administrator's Guide*.

Running the Notification Service

Starting the service

Like all Orbix services, the notification service can be configured to start on demand, to start at system boot, or be started by a script generated by the configuration tool.

You can also manually start the notification service with the following command:

```
itnotify
```

Stopping the Service

To stop the notification service you can use the stop script generated by the configuration tool or you can use the following command:

```
itnotify stop
```

Using Direct Persistence

Overview

By running in direct persistence mode, the notification service can function as a stand-alone component. It does not require the Orbix infrastructure.

Technical Details

When the notification service runs in direct persistence mode it listens on a fixed host and port number. This information is embedded into the IOR that the service exports as an initial reference.

When a CORBA client asks for the notification service's initial reference, it receives the IOR containing the host and port information for the service. The client uses the embedded information to directly contact the notification service, bypassing the locator and node daemon normally used by Orbix CORBA services.

Performance Issues

While direct persistence liberates the notification service from the Orbix infrastructure, it also has a cost in terms of fault tolerance and flexibility. When running in direct persistence mode the notification service cannot be started on demand and must always listen on the configured host and port number.

Configuring Direct Persistence

To configure the notification service to run in direct persistence mode complete the following steps:

1. If the notification service is running, shut it down with the command

```
itnotify stop
```

2. Set `plugins:notify:direct_persistence` to `TRUE` within the notification service's configuration scope. The default scope is `iona_services.notify`.

> **Note:** For information on changing configuration variables, see the *CORBA Administrator's Guide*.

3. Within the same configuration scope, set `plugins:notify:iiop:port` to some open port number.

4. Prepare the service, by running the command

```
itnotify prepare
```

This command causes the notification service to generate a new IOR for itself. The new IOR will be printed to the console. Save it for use in the next step.

5. Within the same configuration scope as used in steps 2 and 3, replace the value of `initial_references:NotificationService:reference` with the IOR returned in step 4.

6. Start the service using the command

```
itnotify
```

Managing a Deployed Notification Service

Using the notification service console

The notification service console provides administrators the ability to monitor and control a deployed notification service. It provides controls to create and destroy notification channels, admin objects, proxy objects, and filters. It also provides controls to edit QoS properties and assign filters and subscriptions to objects in a deployed notification service.

To start the notification console use the following command:

```
itnotify_console
```

The console has detailed context sensitive help to guide you in using it.

Example 1: Generating Trace Information

Scenario

Your company recently installed an inventory control program using Orbix notification to facilitate communication between the sales, manufacturing, and purchasing departments. The sales department takes orders on PDAs and syncs them with the inventory and ordering system when they return to the office. The sales information triggers manufacturing jobs, which in turn produce materials requisitions. The inventory system checks the requisitions against what is in-stock. If all of the requisitions for a job can be filled, the requisitions are filled. If a requisition cannot be filled, the system alerts purchasing and the remaining requisitions are filled.

Problem

A large number of jobs are being held up because the needed materials are not being ordered. After looking for human causes and finding none, the company tasks you with finding the bottleneck in the new system.

Solution

The first step in your task is to determine if the purchasing system is receiving the alert that it needs to order new materials. To accomplish this task you need to turn on the notification services logging facility.

The logging facility is controlled using the variables in the `plugins:notify:trace` namespace. By default they are set to `0`, which means no logging information is generated. To trace events as they pass through the notification service, use `itadmin` to set `plugins:notify:trace:events` to `1`. If you need more detailed information, set the value higher.

Example 2: Failure Recovery

Scenario

Your bank has just converted its ATM network to a system built using Orbix notification. Because of the sensitivity of the information and the fact that it processes information when service personnel may not be immediately available, the system needs to be extremely fault tolerant.

Solution

To increase the fault tolerance of Orbix notification you can change the settings of the variables in the `plugins:notify:database` namespace. These variables control the database used by persistent channels in a deployed notification service.

For example, if you wanted to retain archive copies of old checkpoint logs, you would set `plugins:notify:database:checkpoint_archive_old_files` to `true`. You could also reduce the interval between database checkpoints by setting `plugins:notify:database:checkpoint_interval` to a smaller number.

Part 3

The Telecom Log Service

In this part

This part contains the following chapters:

Telecom Log Service Basics

The telecom log service provides a mechanism for creating a persistent log of events in a distributed computing environment. It provides tools for reviewing past events and it also allows for the recovery of events in the event of a catastrophic failure.

In this chapter

This chapter discusses the following topics:

Telecom Log Service Objects

BasicLog

BasicLog objects provide standard, event-unaware, CORBA objects write access to the telecom log service's persistent store. The BasicLog object can also query the service's persistent store.

EventLog

EventLog objects provide event functionality to event-aware CORBA objects. The EventLog object can forward events from an event supplier to an event consumer. It also allows log clients to receive log generated events.

NotifyLog

NotifyLog objects extend the functionality of the EventLog objects to take advantage of the notification service's filtering and QoS capabilities. NotifyLog objects can also filter the types of events that are logged to the persistent store. You must have a licensed and functioning notification service to use NotifyLog objects.

Factory objects

Each type of log object also has an associated log factory object for creating and managing log objects.

Telecom Log Service Features

Overview

Table 4 shows the features that each type of log object supports.

Table 4: *Log feature support*

Type of Log	Write Operations	Filtering	Event Forwarding	Event Generation	QoS
BasicLog	Store data directly to the log.	None	None	None	Log level QoS
EventLog	Write data directly to the log and push/pull style events.	None	Supports push and pull style forwarding of unstructured events.	Yes	Log level QoS
NotifyLog	Write data directly to the log and push/pull style writing of structured and unstructured events.	Supports filtering of events being written to the log as well as notification style event filtering.	Supports push and pull style forwarding of structured and unstructured events.	Yes	Log level and notification service levels of QoS

Quality of Service

The telecom log service offers three quality of service levels:

QoSNone specifies that log records are buffered in memory when received and are written to the persistent store by the log at preconfigured intervals.

QoSFlush specifies that log records are buffered in memory and are written to the persistent store when the `flush()` method is invoked on the log object.

QosReliability specifies that log records are written directly to the persistent store.

Developing Telecom Log Clients

Clients connect to the telecom log service to create a persistent record of their activities.

In this chapter

This chapter discusses the following topics:

Creating a Log

Overview

The telecom log service provides a factory object for each type of logging object. A factory object, which also acts as a manger for the log objects it creates, can be used to instantiate log objects of the same type. For example, a `NotifyLogFactory` object would be used to instantiate a `NotifyLog` object.

Steps

To create a log object complete the following steps:

1. "Obtain a log factory"
2. "Obtain a log object"

Obtain a log factory

You obtain a log factory by resolving the telecom log service's initial reference through the ORB, by calling `resolve_initial_references()` with the string for the type of log factory you wish to obtain.

Table 5 lists the string to use for each factory object.

Table 5: *Initial reference strings*

Factory	Initial Reference String
BasicLogFactory	BasicLoggingService
EventLogFactory	EventLoggingService
NotifyLogFactory	NotifyLoggingService

Once you have obtained the object reference from `resolve_initial_references()`, you need to narrow it to the proper object type (`BasicLogFactory`, `EventLogFactory`, or `NotifyLogFactory`).

Example 32 shows how to obtain the `NotifyLogFactory`.

Example 32: *Obtaining a NotifyLogFactory*

```
// Java
import org.omg.DsNotifyLogAdmin.*;

org.omg.CORBA.ORB orb = org.omg.CORBA.ORB_init(args, props);
org.omg.CORBA.Object obj =
    orb.resolve_initial_references("NotifyLoggingService");
NotifyLogFactory factory = NotifyLogFactoryHelper.narrow(obj);
```

The lines labeled **1**, **2**, **3** correspond to:

1. Initialize the orb.
2. Obtain a reference to the `NotifyLoggingService`.
3. Narrow the object reference to the `NotifyLogFactory`.

Obtain a log object

Overview

Once you have a log factory, you can then obtain a log object from it. The log factories provide three methods of obtaining a log object:

find_log() allows you to find a log object using its unique id number.

create() creates a log object with an id assigned by the telecom log service.

create_with_id() creates a log object with a user assigned id.

Finding a log

If you have a specific log object you wish to use and you know its id, you can call the log factory's `find_log()` method. It has the following syntax:

```
// IDL
Log find_log(in LogId id);
```

If the log exists `find_log()` returns a reference to the log object. Otherwise, it returns a nil object reference.

Creating a BasicLog

A `BasicLog` object is created from the `BasicLogFactory`. Once you have obtained the `BasicLogFactory` from the ORB, you can use either the `create()` method or `create_with_id()` method to create a `BasicLog`.

create()

The `BasicLogFactory`'s `create()` method has the following signature:

```
// IDL
BasicLog create(in LogFullActionType  full_action,
                in unsigned long long max_size,
                out LogId             id)
raises (InvalidLogFullAction);
```

It takes the following parameters:

full_action defines how the log will behave once it has reached it maximum size. Table 6 shows the possible values for full_action.

Table 6: *Settings for a log's full_action*

Value	Behavior
halt	The log stops logging events until the old events have been cleared out and the log's size is below its max size.
wrap	The log will wipe out the oldest events to make room for new event logging.

max_size specifies the maximum size of the log in bytes.

id is the unique id assigned to the log object by the log factory.

create() will raise the InvalidLogFullAction exception if full_action is not a valid LogFullActionType.

create_with_id()

The BasicLogFactory's create_with_id() method has the following signature:

```
// IDL
BasicLog create_with_id(in LogID              id
                        in LogFullActionType  full_action,
                        in unsigned long long max_size,
raises (LogIdAlreadyExists, InvalidLogFullAction);
```

It takes the following parameters:

id is the log object's unique id.

full_action defines how the log will behave once it has reached it maximum size. Table 6 on page 167 shows the possible values for full_action.

max_size specifies the maximum size of the log in bytes.

create_with_id() raises the following exceptions:

LogIdAlreadyExists is raised if a log object is already using the id you passed as a parameter.

InvalidLogFullAction is raised if `full_action` is not a valid `LogFullActionType`.

Creating an EventLog

An `EventLog` object is created from the `EventLogFactory`. Once you have obtained the `EventLogFactory` from the ORB, you can use either the `create()` method or `create_with_id()` method to create an `EventLog`.

When a new log object is created, the `EventLogFactory` generates an `ObjectCreation` event.

create()

The `EventLogFactory`'s `create()` method has the following signature:

```
// IDL
EventLog create(in LogFullActionType          full_action,
                in unsigned long long         max_size,
                in CapacityAlarmThresholdList thresholds,
                out LogId                     id)
raises (InvalidLogFullAction,
        InvalidThreshold);
```

The `EventLogFactory`'s `create()` method is similar to the `BasicLogFactory`'s `create()` method. See "create()" on page 166. However, the `EventLogFactory` adds the `thresholds` parameter. This parameter holds a sequence of `short` which specifies, as a percentage of max log size, the points at which an `ThresholdAlarm` event will be generated. If an invalid threshold value is passed to the method, `InvalidThreshold` exception is thrown.

create_with_id()

The `create_with_id()` method also takes the additional `thresholds` parameter and will throw `InvalidThreshold`. Otherwise it is identical to the `BasicLogFactory`'s `create_with_id()` method. See "create_with_id()" on page 167.

Creating a NotifyLog

A `NotifyLog` object is created from the `NotifyLogFactory`. Once you have obtained the `NotifyLogFactory` from the ORB, you can use either the `create()` method or `create_with_id()` method to create a `NotifyLog`.

create()

The `NotifyLogFactory`'s `create()` method has the following signature:

```
// IDL
NotifyLog create(in DsLogAdmin::LogFullActionType full_action,
             in unsigned long long max_size,
             in DsLogAdmin::CapacityAlarmThresholdList thresholds,
             in CosNotification::QoSProperties initial_qos,
             in CosNotification::AdminProperties initial_admin,
             out DsLogAdmin::LogId id)
  raises(DsLogAdmin::InvalidLogFullAction,
         DsLogAdmin::InvalidThreshold,
         CosNotification::UnsupportedQoS,
         CosNotification::UnsupportedAdmin);
```

The `NotifyLogFactory`'s `create()` method extends the functionality of the `EventLogFactory`'s `create()` method by including parameters to support a Notification Channel. These parameters are:

initial_qos specifies the initial QoS properties for the log's associated notification channel.

initial_admin specifies the initial admin properties for the log's associated notification channel.

id is the unique id assigned to the log object by the log factory.

create_with_id()

The `NotifyLogFactory`'s `create_with_id()` method has the following signature:

```
// IDL
NotifyLog create_with_id(in DsLogAdmin::LogId id,
             in DsLogAdmin::LogFullActionType full_action,
             in unsigned long long max_size,
             in DsLogAdmin::CapacityAlarmThresholdList thresholds,
             in CosNotification::QoSProperties initial_qos,
             in CosNotification::AdminProperties initial_admin)
  raises(DsLogAdmin::LogIdAlreadyExists,
         DsLogAdmin::InvalidLogFullAction,
         DsLogAdmin::InvalidThreshold,
         CosNotification::UnsupportedQoS,
         CosNotification::UnsupportedAdmin);
```

When a new log object is created, the `NotifyLogFactory` generates an `ObjectCreation` event.

The `NotifyLogFactory`'s `create()` and `create_with_id()` methods are similar to the `EventLogFactory`'s `create()` and `create_with_id()` methods. See "Creating an EventLog" on page 168. However, the `NotifyLogFactory` inherits the `CosNotifyChannelAdmin::ConsumerAdmin` interface and `NotifyLog` objects take full advantage of the telecom log service's ability to provide notification channel functionality. Therefore, the `NotifyLogFactory`'s `create()` and `create_with_id()` methods have two additional parameters. One configures its QoS properties and one configures its Admin properties. In addition, the `NotifyLogFactory`'s methods throw both the `CosNotification::UnsupportedQoS` exception and the `CosNotification::UnsupportedAdmin` exception. For more information see "Notification Service Properties" on page 57.

Example

Example 33 creates a `NotifyLog` of type `QoSNone` that will generate a `ThresholdAlarm` when it reaches 90% of its maximum capacity. Note that by default, the log will be created with the `QoSNone` QoS property.

Example 33: *Creating a NotifyLog*

```java
// Java
import org.omg.DsLogAdmin.*
import org.omg.CosNotification.*;

IntHolder id = new IntHolder();
1   // create the notification QoS properties
Property[] qos = new Property[0];

// create the notification Admin properties
Property[] admin = new Property[0];
2   // Set a threshold alarm at 90% full
short[] threshold = new short[1];
threshold[0] = 90;
3   // factory obtained previously
NotifyLog log = factory.create(halt, 0, threshold, qos, admin,
    id);
```

1. Specify the QoS and Admin properties for the log object.
2. Specify the threshold list for the log object.
3. Call `create()` on the factory object to get the log object.

Logging Events

Overview

Events are stored in the log's persistent database. This is accomplished by one of two mechanisms, for `BasicLog` objects you must directly call the `write_records()` method or the `write_recordlist()` method. You can use the `write_records()` or the `write_recordlist()` methods to write data directly to the log. In addition to using `write_records()` and `write_recordlist()`, you can record events in `EventLog` and `NotifyLog` objects using the push/pull mechanisms as you would when using the event or notification service.

When data is recorded in the log, it is assigned a unique id and the time it was recorded is noted. This information is stored in a record header that can be used to retrieve the data.

In this section

This section discusses the following topics:

Logging with a BasicLog

Overview

BasicLog objects have no knowledge of events or event channels and therefore must communicate directly with the log. The write_records() method and a write_recordlist() method, specified in DsLogAdmin::Log, provide BasicLog objects with this functionality.

write_records()

write_records() has the following signature:

```
// IDL in DsLogAdmin::Log
typedef sequnce<any> Anys;

void write_records(in Anys records)
raises(LogFull, LogOffDuty, LogLocked, LogDisabled);
```

It takes a sequence of Any that contains the data to be logged. The data is recorded directly into the log without any filtering or indexing. It raises the following exceptions:

LogFull	Raised if the log has reached its maximum size and its full action is set to halt.
LogOffDuty	Raised when the log is not scheduled to receive data.
LogLocked	Raised when the log's administrative state is set to locked.
LogDisabled	Raised when the log's operational state is set to disabled.

To store data using write_records() complete the following steps:

1. Package the data to be logged into a DsLogAdmin::Anys, which is a sequence of Any.
2. Invoke write_records() on the log.
3. Catch any exceptions.

Example 34 writes a record containing information about a cell phone call. The information logged is the number the call originated from, the number called, and the reason for the event.

Example 34: *Writing data to a BasicLog object*

```java
// Java
import org.omg.DsLogAdmin.*;

DsLogAdmin.Any[] anys = new DsLogAdmin.Any[3];
any[0].value.insert_string("7989028321");
any[1].value.insert_string("8606531000");
any[2].value.insert_string("connected");

try
{
  log.write_records(anys);  // log obtained earlier
}
catch(const DsLogAdmin::LogFull&)
{
  System.err.println("'Basic log "+log->id()+"' is full");
}
catch(const DsLogAdmin::LogOffDuty&)
{
  System.err.println("'Basic log "+log->id()+"' is off duty");
}
catch(const DsLogAdmin::LogLocked&)
{
  System.err.println("'Basic log "+log->id()+"' is locked");
}
catch(const DsLogAdmin::LogDisabled&)
{
  System.err.println("'Basic log "+log->id()+"' is disabled");
}
```

The numbers **1**, **2**, **3** appear in the left margin aligned with the corresponding code lines.

write_recordlist()

write_recordlist() has the following signature:

```
// IDL is DsLogAdmin.idl
struct NVPair
{
  string name;
  any    value;
};

typedef sequence<NVPair> NVList;

struct LogRecord
{
  RecordId id;
  TimeT    time;
  NVList   attr_list; // attributes, optional
  any      info;
};
typedef sequence<LogRecord> RecordList;

void write_recordlist(in RecordList list)
raises(LogFull, LogOffDuty, LogLocked, LogDisabled);
```

write_recordlist() is functionally identical to write_records(). It writes data directly to the log and raises the same exceptions. The major difference is that the record's data is stored in a LogRecord. This allows you to add a series of name/value pair attributes to assist in querying the log.

To store data using write_recordlist() complete the following steps:

1. Package the data to be logged into a DsLogAdmin::RecordList, which is a sequence of LogRecord. Each record's id and time members will be filled in by the log.

2. Invoke write_recordlist() on the log.

3. Catch any exceptions.

Example 35 writes a record to a BasicLog object using write_recordlist(). The record includes a single attribute that identifies the type of minutes being billed.

Example 35: *Writing data to a* `BasicLog` *object*

```
// Java
import org.omg.DsLogAdmin.*;
```

1
```
//Create a new log record
LogRecord record = new LogRecord();

// create a new attribute list
record.attr_list[] = new NVList[1];
record.attr_list[0].name = "minute_type";
record.attr_list[0].value =
   org.omg.CORBA.ORB.init().create_any();
record.attr_list[0].value.insert_string("free");

// Load the data into an any
record.info = org.omg.CORBA.ORB.init().create_any();
record.info.insert_string("7989028321, 8606531000, connected");

// Create a RecordList
Record[] records =new Records[1];
records[0] = record;
```

2
```
try
{
  log.write_recordlist(records);  // log obtained previously
}
```
3
```
catch(const DsLogAdmin::LogFull&)
{
  System.err.println("'Basic log "+log->id()+"' is full");
}
catch(const DsLogAdmin::LogOffDuty&)
{
  System.err.println("'Basic log "+log->id()+"' is off duty");
}
catch(const DsLogAdmin::LogLocked&)
{
  System.err.println("'Basic log "+log->id()+"' is locked");
}
catch(const DsLogAdmin::LogDisabled&)
{
  System.err.println("'Basic log "+log->id()+"' is disabled");
}
```

Logging Events with an EventLog

Overview

While an `EventLog` object can use the `write_records()` method or the `write_recordlist()` method to log data in a persistent data store, `EventLog` objects also take advantage of the CORBA event services push/pull mechanisms to log events.

Procedure

The procedure for logging events using an `EventLog` object is identical to sending an event through the event service. The object generating the event is an event service supplier and it either pushes events to the log or allows the log to pull events from it depending on the suppliers implementation.

The `EventLog` inherits from the `CosEventChannelAdmin::EventChannel` interface, thus it has the associated methods to connect an event supplier through a proxy consumer.

To log events using an `EventLog`, complete the following steps:

1. Obtain a `SupplierAdmin` from the log.

2. Obtain a proxy consumer from the `SupplierAdmin`.

3. Connect the proxy consumer to the log's event channel.

4. Send events to the log using either `push()` or `pull()` depending on the type of supplier you choose to use.

For more information on connecting supplier to an event channel, see the chapter on the event service in the *CORBA Programmer's Guide*.

Once the supplier is connected to the log, you can continue to pass events to the log until you explicitly disconnect from the log.

Example

Example 36 logs events to an `EventLog` using a push supplier. The code is labeled according to the steps outlined in the procedure above.

Example 36: *Logging events to an EventLog using a push supplier*

```
1   org.omg.CosEventChannelAdmin.SupplierAdmin sa =
        log.for_suppliers();
2   org.omg.CosEventChannelAdmin.ProxyPushConsumer ppc =
        sa.obtain_push_consumer();
3   ppp.connect_push_supplier(org.omg.CosEventComm.PushSullplier._ni
        l());
```

Example 36: *Logging events to an EventLog using a push supplier*

```
org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
any.insert_string("7989028321, 8606531000, connected");
ppc.push(any);
```

4

In step 3 a nil supplier reference is used because the log object does not
need a disconnect notification.

Logging Events with a NotifyLog

Overview

NotifyLog objects are similar to EventLog objects in that they use an event channel and use the push/pull methods to log data. However, NotifyLog objects also inherit from CosNotifyChannelAdmin, which enables them to log sturctured events and sequenced events.

Procedure

The procedure for connecting to a NotifyLog and logging events is the same as that used for a connecting to the notification service.

To log events using an NotifyLog, complete the following steps:

1.   Obtain a SupplierAdmin from the log.

2.   Obtain a proxy consumer from the SupplierAdmin.

3.   Connect the proxy consumer to the log's event channel.

4.   Send events to the log using either push() or pull() depending on the type of supplier you choose to use.

For information on connecting to the notification service, see the "Implementing a Supplier" on page 27.

Example

Example 37 logs events to a NotifyLog using a push supplier.

Example 37: *Logging events to a NotifyLog using a push supplier*

```
// Java
import org.omg.CosNotifyChannelAdmin.*;

1   SupplierAdmin sa = log.default_supplier_admin();

2   IntHolder proxy_id = new IntHolder();
    ClientType ctype = CosNotifyChannelAdmin.ANY_EVENT;
    ProxyConsumer obj = sa.obtain_notification_push_consumer(ctype,
        proxy_id);

3   ProxyPushConsumer pc = ProxyPushConsumerHelper.narrow(obj);

4   pc.connect_any_push_supplier(CosEventComm.PushSupplier._nil());
```

Example 37: *Logging events to a NotifyLog using a push supplier*

```
org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
any.insert_string("7989028321, 8606531000, connected");

pc.push(any);
```

5

1. Get the default `SupplierAdmin` object for the log's notification channel.
2. Get a proxy consumer that uses unstructured events.
3. Narrow the returned proxy to a `ProxyPushConsumer`.
4. Connect the proxy consumer to the log's notification channel. A nil reference can be passed because the log does not need to be notified of a disconnect.
5. Push the event to the log.

Getting Log Records

Overview

When a record is stored in the log, the log creates a header for it that contains a unique id for the record and the time that the data was recorded. This header can also contain an optional attribute list. Using this data, you can retrieve records from the log.

The telecom log service provides two methods for getting records from the log:

1. You can retrieve a series of records based on the time when they were logged. For example, you can retrieve the first 100 records logged after 10pm February 3, 2002.

2. You can retrieve records based on a search criteria. For example, you can retrieve all of the events that record losses by your local rugby team.

Retrieving records based on time logged

The `retrieve()` operation reads the log records in the log sequentially starting from any given time. It has the following signature:

```
// IDL
RecordList retrieve(in TimeT start, in long num, out Iterator i);
```

If a negative value for the number of records to retrieve is supplied, `retirieve()` will return records that were logged prior to the start time, starting with the most recently logged and ending with the oldest in the series

The iterator value is used to handle the retrieval of large amounts of data. If the number of records specified cannot fit in the return value, the iterator provides access to the remaining records. If the iterator is not needed it will be nil.

Querying the log for records

Each log record contains the time it was logged, a unique record id, a set of optional attributes, and the data being logged. Queries can be constructed to retrieve log records based on any of this information.

Queries are constructed using a constraint language based on the standard OMG Trader Constraint Language with some modifications that make it

more suitable for use in querying log records. For more information on the constraint language, see "Filter Constraint Language" on page 100.

The `query()` operation takes in a constraint and returns all of the records in the log that matches it. `query()` has the following signature:

```
// IDL
RecordList query(in string grammar, in Constraint match_string,
    out Iterator i)
raises(InvalidGrammar, InvalidConstraint);
```

The grammar parameter indicates how to interpret the constraint string. The default grammar is "EXTENDED_TCL". The records which match the constraint, `match_string`, are returned as a `RecordList`. An iterator may be returned to handle large query results. A nil object reference will be returned for the iterator if it is not needed.

`query()` can raise the following exceptions:

InvalidGrammar   Raised if the log does not support the grammar specified.

InvalidConstraint Raised if the constraint string is invalid.

Example 38 retrieves all of the records that have the attribute `minute_type` set to "`roaming`".

Example 38: *Querying a log for records*

```
// Java
org.omg.DsLogAdmin.IteratorHolder iter = new
    org.omg.DsLogAdmin.IteratorHolder();

org.omg.DsLogAdmin.Record[] list = log.querey("EXTENDED_TCL",
    "$minute_type == 'roaming'", iter);
```

Deleting Records from the Log

Overview

Records are removed from the log automatically once they reach their life expectancy. However, it is occasionally necessary to delete records from the log. The telecom log service provides you with the option of deleting specific records based on their record id or deleting records based on a constraint.

Deleting records by id

The `delete_records_by_id()` operation deletes specific log records from the log. It takes a sequence of `RecordId` as a parameter, and returns the number of records deleted. If no records match the ids specified, the operation will return 0.

Using a constraint to delete records

The `delete_records()` operation deletes records from the log based on a constraint. See "Querying the log for records" on page 180 for more information on how to form a constraint.

It returns the number of records deleted and can raise the following exceptions:

`InvalidGrammar`   Raised if the implementation does not support the specified grammar.

`InvalidConstraint` Raised if the constraint string is invalid.

`InvalidAttribute` Raised if one of the attributes specified in the constraint string is invalid.

Example 39 deletes all of the records whose id is less than 10.

Example 39: *Deleting records from a log*

```
// C++
CORBA::ULong deled = log->delete_records("EXTENDED_TCL", "$.id <
    10");
cout << deled << "records deleted from the log." << endl;
```

Example 40: *Deleting records from a log*

```
// Java
org.omg.CORBA.Ulong deled = log.delete_records("EXTENDED_TCL",
    "$.id < 10");
System.out.println(deled + " records deleted from the log.");
```

Ending a Logging Session

Overview

To end a logging session, the client needs to release the object reference to the log object. For `EventLog` objects and `NotifyLog` objects, the developer must also disconnect the client from the event channel associated with the log.

Using the `destroy()` operation will eliminate the object instantiating the log in the telecom log service and destroy any records stored in the log.

Advanced Features

The telecom log service provides a number of features to make it flexible enough to handle most enterprise level applications. Most of the features leverage the functionality of the event and notification services and are therefore only available to EventLogs and NotifyLogs.

In this chapter

This chapter discusses the following topics:

Scheduling

Overview

All log implementations allow you to schedule when the log is active. During this time, it will be fully functional and log messages. When the log is not scheduled to log new records, it will still be available for record retrieval and event forwarding.

Scheduling scenario

The ability to schedule when the log records data can be valuable to control both the size of the persistent store and the overall performance of your system. For example, suppose you need to develop an application to monitor the performance of a cell phone network. During peak hours, there are millions of events generated per hour on the network and there are technicians on hand at all times. During off-peak hours, the number of events generated is cut in half and there is only a skeleton crew of technicians available to handle critical failures.

The added overhead of logging events during peak hours will most likely have serious implications in overall system performance and may, during particularly heavy periods, be prohibitive. Because there are a number of technicians and support personnel on hand to monitor the network manually, it may not be necessary to log events during peak hours. Therefore you could schedule the log to only log events during off-peak hours when the overhead would be lower and there are not enough technicians to constantly monitor the network.

Schedule data

Log schedules are specified using a `WeekMask` which is a `struct` defined in module `DsLogAdmin`.

```
// IDL in DsLogAdmin
struct Time24
{
    unsigned short hour;    // 0-23
    unsigned short minute; // 0-59
};

struct Time24Interval
{
    Time24 start;
    Time24 stop;
};
typedef sequence<Time24Interval> IntervalsOfDay;

const unsigned short Sunday    =  1;
const unsigned short Monday    =  2;
const unsigned short Tuesday   =  4;
const unsigned short Wednesday =  8;
const unsigned short Thursday  = 16;
const unsigned short Friday    = 32;
const unsigned short Saturday  = 64;

typedef unsigned short DaysOfWeek;// Bit mask of week days

struct WeekMaskItem
{
  DaysOfWeek      days;
  IntervalsOfDay intervals;
};
typedef sequence<WeekMaskItem> WeekMask;
```

The `intervals` field of a `WeekMaskItem` specifies the time, in 24 hour format, that the log will begin logging records and the time that the log will stop logging records.

The `days` field of `WeekMaskItem` indicates which days of the week to apply the `start` and `stop` times specified in the `intervals` field. It is created using a bitwise OR operation to create a bitmask specifying the days. For example, to specify that an interval should be valid on Friday, Saturday, and Sunday you would use the following code:

```
DaysOfWeek days = Friday | Saturday | Sunday;
```

Setting a schedule

By default, a log has no set schedule and will log records continuously. If you want to alter that behavior, you use the set_week_mask() operation to set a schedule for the log. The operation has the following signature:

```
\\ IDL
void set_week_mask(in WeekMask masks)
raises (InvalidTime, InvalidTimeInterval, InvalidMask);
```

The masks parameter allows you to specify as complex a schedule as needed. For instance you can set a different logging interval for each day of the week or specify multiple intervals during a single day to log records (providing the intervals do not overlap).

When using an EventLog or a NotifyLog, an AttributeValueChange event is generated whenever the log's schedule is changed. See "Log Generated Events" on page 190 for more information.

set_week_mask() raises the following exceptions:

InvalidTime	One of the values specified for a start or stop time is not within the valid range.
InvalidTimeInterval	One of the time intervals is improperly formed. For example, the stop time is before the start. Also raised if the intervals overlap.
InvalidMask	The days parameter is malformed.

Example 41 tells a log to log records from 12am until 8am and from 7:30pm until 11:59pm Monday through Friday.

Example 41: *Setting a logs schedule*

```
// Java
import org.omg.DsLogAdmin.*;
```

Example 41: *Setting a logs schedule*

```
// Contruct the times between the log is to record data
IntervalsOfDay[] intervals = new IntervalOfDay[2];
intervals[0].start.hour = 0;
intervals[0].start.minute = 0;
intervals[0].stop.hour = 8;
intervals[0].stop.minute = 0;
intervals[1].start.hour = 19;
intervals[1].start.minute = 30;
intervals[1].stop.hour = 23;
intervals[1].stop.minute = 59;

// Build the mask to specify the days on which
// the schedule is valid
DaysOfWeek days = Monday | Tuesday | Wednesday | Thursday |
   Friday;

// Package the schedule
WeekMask[] sched = new WeekMask[1];
sched[0].days = days;
sched[0].intervals = intervals;

// Apply the schedule to the log
try
{
  log->set_week_mask(sched);
}
// Handle any exceptions
catch(const InvalidTime&)
{
 ...
}
catch(const InvalidTimeInterval&)
{
...
}
catch(const InvalidMask&)
{
...
}
```

Determining a log's schedule

You can determine what schedules, if any, have been set for a given log by calling the get_week_mask() method on it. get_week_mask() takes no parameters and returns the scheduling information for the log in a WeekMask.

189

Log Generated Events

Overview

EventLogFactory and NotifyLogFactory objects can keep their clients informed of the telecom log service's state by generating events and forwarding the events onto their clients. This feature can be particularly useful for developing clients that need to respond gracefully to log failures or other status changes.

For example, you need to implement a system to process purchases made through your companies web site and you decide to use the telecom log service to create a persistent record of the purchases made outside of normal business hours, so that the orders can be handled the following business day. If the log being used to store the purchases reached its threshold before the new purchases could be processed, the log would have two options of how to react, depending on how you set its full_action. The log could either stop recording the purchases, or it could write over the old records. Neither option is acceptable.

If you developed a client that received log generated events, you could design it to handle a full log gracefully. For instance, you could have the client stop accepting new purchases until the log was emptied or you could have it create a new log object and begin to record purchases there.

Log events

Log objects generate events for the following reasons:

Table 7: *Events generated by a log factory*

Event	Reason
ObjectCreation	Generated when a log object is created.
ObjectDeletion	Generated when a log object is destroyed.
ThersholdAlarm	Generated when a log object's threshold capacity is reached. Alarms can be configured at different percentages of the logs capacity. For example, one alarm event can be generated when the log reaches 90% of capacity and another can be generated when the log reaches 95% of capacity.

Table 7: *Events generated by a log factory*

Event	Reason
AttributeValueChange	Generated when a log changes one of the following log attributes: • capacity alarm threshold • log full action • maximum log size • start time • stop time • week mask • adding/removing/changing a constraint expression on the log's filter object • max record life • quality of service
StateChange	Generated when a log object's operational or administrative state is changed.
ProcessingErrorAlarm	Generated when a log generates an error.

Event propagation

The EventLogFactory and NotifyLogFactory interface inherit from the CosEventChannelAdmin::ConsumerAdmin and the CosNotifyChannelAdmin::ConsumerAdmin interfaces, respectively. Therefore event service consumers, both push and pull style, can connect to an EventLogFactory to receive log generated events. Also, notification service consumers, both push and pull, can connect to a NotifyLogFactory to receive log generated events. For more information about event propagation see "Event Communication" on page 8.

Receiving log generated events

To develop a telecom log service client that receives log generated events from the EventLogFactory or the NotifyLogFactory complete the following steps:

1. Obtain a reference to the log factory, either EventLogFactory or NotifyLogFactory. See "Obtain a log factory" on page 165.

2. Obtain a proxy supplier from the log factory.

3. Connect to the proxy supplier using its connect method.

4. For a pull consumer, call `pull()` or `try_pull()` to receive events. For a push consumer, you will need to implement the appropriate `push()` method.

For a more detailed description of how to connect an event consumer to an event channel, see the *CORBA Programmer's Guide* and "Implementing a Consumer" on page 44.

Example 42 implements a push consumer that receives events from the `NotifyLogFactory`.

Example 42: *Receiving events from the NotifyLogFactory*

```
// Java
import org.omg.CORBA.*;
import org.omg.CosNotification.*;
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.CosNotifyComm.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POAPackage.*;
```

1
```
class NotifyPushConsumer extends PushConsumerPOA
{
// member variables not shown...

  void public push(Any event)
  {
    // Process the event
    // ...
  }

  void public NotifyPushConsumer()
    {
    {

  // client consumer program
  public static main(String args[])
    {
      // ORB and POA activation not shown
      // ...
```

2
```
      // Create the push consumer
      NotifyPushConsumer consumer = new NotifyPushConsumer();
```

Example 42: *Receiving events from the NotifyLogFactory*

```
3        // get a reference to the NotifyLogFactory
         Object obj =
      orb.resolve_initial_references("NotifyLoggingService");
         org.omg.DsNotifyLogAdmin.NotifyLogFactory factory =
      org.omg.DsNotifyLogAdmin.NotifyLogFactoryHelper.narrow(obj);

         // The client consumes events of type ANY
         ClientType type = CosNotifyChannelAdmin.ANY_EVENT;

         // get the push proxy supplier
         IntHolder proxy_id = new IntHolder();

4      try
           {
            ProxySupplier obj =
      factory.obtain_notification_push_supplier(type, proxy_id);
           }
        catch(CosNotifyChannelAdmin::AdminLimitExceeded err)
           {
           // handle the exception
           }

        ProxyPushSupplier_var pps =
        ProxyPushSupplierHelper.narrow(obj);

5      try
           {
           pps.connect_push_consumer(consumer)
           }
        catch (AlreadyConnected ac)
           {
           System.out.println("Already connected to channel.");
           exit (1);
           }
        catch (CORBA::SystemException& se)
           {
           System.out.println("System exception occurred during
        connect.");
           exit(1);
           }

       // ...

     } // main
   } // NotifyPushConsumer
```

1. Implement the consumer's class and its `push()` method.
2. Instantiate the consumer.
3. Obtain a reference to the `NotifyLogFactory`, which inherits from `CosNotifyChannelAdmin::ConsumerAdmin`.
4. Obtain a push supplier from the log factory and narrow it to a `ProxyPushSupplier`.
5. Connect the consumer to its proxy supplier.

Once the consumer is connected to its proxy it will continue to receive log generated events until it explicitly disconnects.

Event data types

Each event generated by the telecom log service is passed to the clients as an any and the clients are responsible for unpacking the data correctly before decoding it. The data types defined for each event provide all of the information necessary to describe the action that generated the event. For example, an `AttributeValueChanged` event's data structure includes a field to describe which attribute was changed, the old value of the attribute, and the new value of the attribute.

ObjectCreation event

An `ObjectCreation` event has the following data structure:

```
// IDL
struct ObjectCreation
{
  LogId id;
  TimeT time;
};
```

It contains the new log's id and the time that the new log was created.

ObjectDeletion event

An `ObjectDeletion` event has the following data structure:

```
// IDL
struct ObjectDeletion
{
  LogId id;
  TimeT time;
};
```

It contains the id of the deleted log and the time it was deleted.

ThresholdAlarm event

A `ThresholdAlarm` event has the following data structure:

```
// IDL
struct ThresholdAlarm
{
  Log logref;
  LogId id;
  TimeT time;
  Threshold crossed_value;
  Threshold observed_value;
  PerceivedSeverityType perceived_severity;
};
```

It contains the object reference and the id of the log whose alarm was set off and the time when the log reached its capacity alarm threshold. The `observed_value` field indicates the log's size, as a percentage of the maximum log size. The `crossed_value` field indicates the threshold level that was crossed. The `perceived_severity` field is `minor` if log is not full, and `critical` otherwise.

AttributeValueChanged event

An `AttributeValueChanged` event has the following data structure:

```
// IDL
struct AttributeValueChange
{
  Log logref;
  LogId id;
  TimeT time;
  AttributeType type;
  any old_value;
  any new_value;
};
```

Along with the affected log's object reference, the affected log's id, and the time of the event, the data structure includes the `type` field which identifies the attribute that was changed, the old value of the attribute, and the new value of the attribute.

StateChange event

A `StateChange` event has the following data structure:

```
// IDL
struct StateChange
{
  Log logref;
  LogId id;
  TimeT time;
  StateType type;
  any new_value;
};
```

Along with the affected log's object reference, the affected log's id, and the time of the event, the data structure includes the `type` field, which identifies the attribute that was changed, and the `new_value` field, which contains the new value of the attribute.

ProcessingErrorAlarm event

A `ProcessErrorAlarm` event has the following data structure:

```
// IDL
struct ProcessingErrorAlarm
{
  long error_num;
  string error_string;
};
```

It contains the error number and a textual description of the log object's error.

Unpacking log generated events

Clients can determine how to unpack log generated events in one of two ways:

Trial and Error

You can code the client code to simply keep trying to stuff the returned `any` into the different log event data structures. Example 43 shows client code

for unpacking log generated events by trial and error.

Example 43: *Unpacking an event by trial and error*

```java
// Java
org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
any = // the event received by the client.

const org.omg.DsLogNotification.ObjectCreation* object_creation;
const org.omg.DsLogNotification.ObjectDeletion* object_deletion;

if(any >>= object_creation)
{
    // An object creation event was received.
}
else if(any >>= object_deletion)
{
    // An object deletion event was received.
}
else
{
    // Some other event type...
}
```

Type Codes

You can also use the type code of the returned `any` to determine what type of event was returned and unpack it accordingly. Example 44 shows client code for unpacking log generated events based on their typecode.

Example 44: *Unpacking log generated events by typecode*

```java
// Java
org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
any = // the event received by the client.

org.omg.CORBA.TypeCode tc = any.type();
if(tc.equivalent(DsLogNotification.ObjectCreationHelper.type()))
{
  // An object creation event was received.
  // Unpack the event and handle the results.
}
else
    if(tc.quivalent(DsLogNotification.ObjectDeletionHelper.type()
    ))
{
  // An object deletion event was received.
  // Unpack the event and handle the results.
}
else
{
    // Some other event type...
}
```

When using `NotifyLog` clients, you can limit the type of events they receive from the log by filtering out the events you do not want the client to receive. See "Filtering" on page 206 and "Event Filtering" on page 83 for detailed information on event filtering.

Event Forwarding

Overview

As seen in Figure 3 on page 6 the telecom log service encapsulates an event channel to provide added functionality to `EventLog` objects and `NotifyLog` objects. Therefore both `EventLog` objects and `NotifyLog` objects are capable of emulating an event channel and passing events between suppliers and consumers using both the push and pull methods. `NotifyLog` clients can also take advantage of the notification service style QoS properties and notification style filtering. See "Filtering" on page 206 and "Log Management" on page 210.

Logs will forward events as long as their `ForwardingState` attribute is set to `on`. Changing a log's administrative state or using a schedule to turn logging on and off does not affect the log's ability to forward events.

The basic steps involved in log event forwarding are:

1. Set the log's `ForwardingState` to `on`. This is the default for all newly created `EventLog` objects and `NotifyLog` objects.

2. Connect the clients to the log object via the event or notification channel interface it supports.

3. `NotifyLog` clients specify filters. See "Filtering" on page 206.

4. Suppliers send events to the log by using either `push()` for push style suppliers, or `pull()` for pull style suppliers. Pull style suppliers can also use `try_pull()`.

5. If the log is set to log events, the events sent to the log object will be recorded.

6. Consumers receive events from the channel.

Developing a telecom log application that uses event forwarding

Developing a telecom log service that uses event forwarding is essentially identical to developing an event service or notification service application. However, the telecom log service has the added benefit that it will maintain a persistent and fully accessible history of the events that are being passed through the channel. The telecom log service suppliers can also be implemented to receive log generated events. See "Log Generated Events" on page 190.

To develop a telecom log service application that forwards events between event suppliers and event consumers complete the following steps:

1. Implement the required methods for the event supplier. If you use a pull style supplier, you will need to implement the appropriate `pull()` and/or `try_pull()` method.

2. Implement the required methods for the event consumer class. If you use a push style consumer, you will need to implement the appropriate `push()` method.

3. Instantiate both the supplier's class and the consumer's class.

4. Obtain either an `EventLog` object or a `NotifyLog` object that has its `ForwardingState` set to `on`.

5. Connect the supplier to the log's associated event channel by obtaining a `SupplierAdmin` from the log object. From the `SupplierAdmin`, you obtain a `ProxyConsumer` to connect to the channel.

6. Begin generating events.

7. Connect the consumer to the log's associated event channel by obtaining a `ConsumerAdmin` from the log object. From the `ConsumerAdmin`, you obtain a `ProxySupplier` to connect to the channel.

For a detailed description of implementing event consumers and event suppliers, see "Developing Suppliers and Consumers" on page 21 and the *CORBA Programmer's Guide*.

NotifyLog features

If you are using a `NotifyLog` object, you can take full advantage of all of the notification services features. These include: event filtering, structured and sequence events, event subscription, and notification-style QoS properties for events. See "Notification Service Properties" on page 57.

Example

The following example implements an application that passes an unstructured event containing the price of a stock from a notification push supplier to a notification push consumer. They both connect to a `NotifyLog` with the id 123. By using a log with a user defined `id`, you ensure that the consumer and the supplier are connected to the log object.

Example 45 implements the notification push supplier.

Example 45: *Implementing the push supplier.*

```java
// Java
import org.omg.DsNotifyLogAdmin.*;
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
import org.omg.CosNotification.*;
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.CosNotifyComm.*;
import org.omg.TimeBase.*;

import org.omg.PortableServer.*;
import org.omg.PortableServer.POAPackage.*;
```

1
```java
class NotifyPushSupplier extends PushSupplierPOA
{
// Member variables not shown...

  public void NotifyPushSupplier()
    {
  // Implementation not shown...
    }
```

2
```java
// The main entry point @param args command line args
    public static void main (String args[])
      {
      // ORB and POA Activation not shown
      NotifyPushSupplier supplier = new NotifyPushSupplier();
```

3
```java
    // Get a Log Factory
    Obect obj =
    orb.resolve_initial_references("NotifyLoggingService");
    NotifyLogFactory factory =
    NotifyLogFactoryHelper.narrow(obj);
```

Example 45: *Implementing the push supplier.*

```
4     // The log will have an id of 123
        IntHolder id = new IntHolder();
        id = (ULong)123;

      // Set the Log's QoS properties
        Property[] qos = new Property[0];
        qos[0].name = Type;
        qos[0].value.insert_int(QoSNone);

        Property[] admin = new Property[0];
        CapacityAlarmThresholdList thresholds = null;

        NotifyLog log = factory.create_with_id(id, halt, 0,
      thresholds, qos, admin);

5       SupplierAdmin sa =   log.default_supplier_admin();

        IntHolder proxy_id = new IntHolder();
        ClientType ctype = ClientType.ANY_EVENT;
        try
          {
          ProxyConsumer obj =
      sa.obtain_notification_push_consumer(ctype, proxy_id);
           }
        catch(AdminLimitExceeded err)
          {
          // handle the exception
          }

        ProxyPushConsumer ppc = ProxyPushConsumerHelper.narrow(obj);
6       Any any = org.omg.CORBA.ORB.init().create_any();
        any.insert_string("FKUSX, $33.02");
        ppc.push(any);

        // ...
      }
   }
```

The supplier code show in Example 45 does the following:

1. Implements the supplier's object class.

2. Instantiates a supplier object.

3. Initializes the ORB and uses `resolve_initial_references()` to get a reference to the `NotifyLogFactory`.

4. Creates a log with an id of 123 using `create_with_id()`. The log is of type `QoSNone` and does not have any threshold alarms set.

5. Obtains a `ProxyPushConsumer` and connects to the log's associated notification channel.

6. Pushes a single event.

Example 46 implements the notification push consumer.

Example 46: *Implementing the push consumer*

```java
// Java
import org.omg.CORBA.*;
import org.omg.CosNotification.*;
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.CosNotifyComm.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POAPackage.*;

class NotifyPushConsumer extends PushConsumerPOA
{
// member variables not shown...

  void public NotifyPushConsumer()
    {
    }

  public void push(Any event)
    {
      if ((event.type()).kind() == TCKind.tk_string)
        {
          String stock_pice = event.extract_string();
          System.out.println("Stock price is" + stock_price);
          }
        else
          System.out.println("Invalid Event");
    }

// The main entry point @param args command line args
  public static void main (String args[])
    {
      // ORB and POA initialization not shown ...

      NotifyPushConsumer consumer = new NotifyPushConsumer();
```

1

2

203

Example 46: *Implementing the push consumer*

```
3        Obect obj =
     orb.resolve_initial_references("NotifyLoggingService");
        NotifyLogFactory factory =
     NotifyLogFactoryHelper.narrow(obj);

4      IntHolder id = 123;
       NotifyLog log = factory.find_log(id))

5      ConsumerAdmin ca = log.default_consumer_admin();

       IntHolder proxy_id = new IntHolder();
       ClientType ctype = ClientType.ANY_EVENT;

       try
         {
         ProxySupplier obj =
     ca.obtain_notification_push_supplier(ctype, proxy_id);
         }
       catch(AdminLimitExceeded err)
         {
          //handle exception
         }

       ProxyPushSupplier pps = ProxyPushSupplierHelper.narrow(obj);

       try
         {
         pps.connect_push_consumer(consumer);
         }
       catch(AlreadyConnected.value ac)
         {
         System.err.println("Already connecting to channel.");
         System.exit(1);
         }
       catch (SystemException sys)
         {
         System.err.println("Encountered system exception during
      connect: " + SystemExceptionDisplayHelper.toString(sys));
         System.exit(1);
         }

6      orb.run();
     }
   }
```

The consumer code show in Example 46 does the following:

1. Implements the consumer's object class.

2. Instantiates a consumer object.

3. Uses `resolve_initial_references()` to get a reference to the `NotifyLogFactory`.

4. Uses `find_log()` to obtain a reference the log created by the supplier.

5. Obtains a `ProxyPushSupplier` and connects to the log's associated notification channel.

6. Turns control over to the ORB to wait for events.

Filtering

Overview

`NotifyLog` objects support two types of filtering:

- Notification style filtering which determines if an event passes through the log's associated event channel.
- Log filtering which determines if an event is logged.

Figure 13 on page 206 shows the different types of filters that can be used by a `NotifyLog`. Notification style filters are applied to the admin and proxy objects in the `NotifyLog` object's associated event channel. Each admin and proxy object may have multiple filters associated with it. If an event is discarded due to a filter on a proxy consumer or supplier admin, it will not reach the log filter and will not be logged.
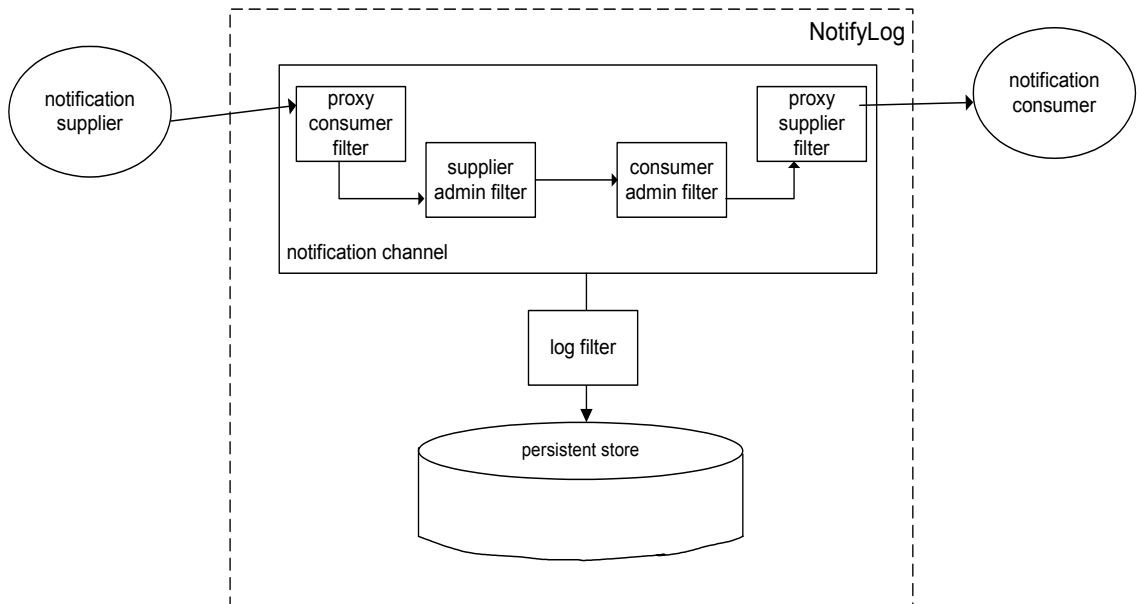
Figure 13: *Filter points in event's life-cycle*

Log filters are applied directly to the log object and do not effect the forwarding of an event. If the event does not pass the log filter, it will not be logged, but it will be passed on to the consumer admin. Unlike a proxy or admin object, a log object can only have one filter associated with it. The log filter can be useful in situations where the log's clients are generating a large number of events of varying types. If you are only interested in a few types of events, you can control the size of the log by applying filters. For example, you can log only events whose **"severity"** is greater than 4 or events with a **"log"** attribute of 1.

For a more detailed discussion of filtering, see "Event Filtering" on page 83.

Implementing a filter

To implement a filter complete the following steps:

1. Obtain a filter factory from the log using the default_filter_factory() method.

2. Create a filter using the factory's create_filter() method. Specify the EXTENDED_TCL grammar, which is the same grammar used by the notification service. See "Filter Constraint Language" on page 100.

3. Build your constraints for the filter, and add them using the filter's add_constraints() method.

4. Attach the filter to the desired object (proxy, admin, or log) using the appropriate method. Table 8 on page 207 shows the method used to attach a filter to the specified object.

Table 8: *Methods for attaching filters*

Object	Method
log object	set_filter(CosNotifyFilter::Filter filter)
proxy object	add_filter(CosNotifyFilter::Filter filter)
admin object	add_filter(CosNotifyFilter::Filter filter)

Example 47 on page 208 creates a filter to log data error events whose
`severity` is greater than 4 and attaches it to the log.

Example 47: *Attaching a filter to a log object*

```
// Java
import org.omg.CosNotifyFilter.*;
import org.omg.CosNotification.*;
```

1
```
// NotifyLog log obtained earlier
FilterFactory dff = log.default_filter_factory();
```

2
```
Filter filter = dff.create_filter("EXTENDED_TCL");
```

3
```
// create a constraint
EventType[] event_types = new EventType(1);

event_types[0].domain_name = new String("Dial Up");
event_types[0].type_name = new String("Data Error");

ConstraintExp[] constraints = ConstraintExp(1);

constraints[0].event_types = event_types;
constraint[0].constraint_expr = new String("$severity > '4'");

ConstraintInfo[] info = filter.add_constraints(constraint);
```

4
```
log.set_filter(filter)
```

Filter evaluation

An event must pass each notification style filter before it is forwarded to the
next point in the channel. If filters are set on an admin object and one of its
proxies, events can be evaluated against both sets of filters, depending on
whether the admin object was created with AND or OR semantics:

- AND semantics require events to pass both admin and proxy filters.
- OR semantics only require an event to pass an admin or proxy filter.

A filter evaluates an event against its set of constraints until one evaluates to
true. A constraint evaluates to true when both of the following conditions are
true:

- A member of the constraint's `EventTypeSeq` matches the message's
  event type.
- The constraint expression evaluates to true.

The first filter in which the event message evaluates to true forwards the event to the next delivery point in the channel. If the event message fails to pass any filters, the event may not be forwarded.

Log Management

Overview

The telecom log service allows you to control the following attributes of a log:

- Administrative State
- Maximum log size
- Log duration
- Record lifetime
- Log QoS properties

You can also monitor a log's availability status, its operational state, and its current size (in bytes and number of records).

In this section

This section discusses the following topics:

Administrative State

Overview

Administrative state can also be thought of as the "logging state" and is used to turn logging on and off. A log's administrative state does not affect the log's ability to forward events. If the administrative state of the log is locked, events will pass through the event channel as long as the log's forwarding state is set to on.

States

Logs can be put into one of two administrative states:

Table 9: *Administrative states for a log*

Administrative State	Log Functionality
Unlocked	The log is fully functional. New records can be added. Records can be retrieved and deleted from the log. Events can be forwarded.
Locked	The log will not create new records. All other functionality of the log is still available.

By default, the administrative state of a newly created log object is unlocked.

Methods

You can determine the administrative state of a log by using its get_administrative_state() method. It returns the administrative state in the enumerated type, AdministrativeState.

You set a log's administrative state using its set_administrative_state() method, which takes a single parameter of type AdministrativeState. A StateChange event is generated whenever the administrative state of a log is changed.

Example

Example 48 checks to see if a log is locked and if it is changes its administrative state to unlocked.

Example 48: *Setting a log's administrative state*

```java
// Java
// log obtained previously
if (log.get_administrative_state() ==
   AdministrativeState.locked)
  {
  log.set_administrative_state(AdministrativeState.unlocked);
  System.out.println("Log " + log.id() + "is now unlocked.");
  }
```

Maximum Log Size

Setting

A log's `set_max_size()` method sets its maximum size in bytes. The method takes an `unsigned long long`. If a value of zero is supplied, then the log size will be set to have no predefined limit. If the new maximum log size is less than the current log size, an `InvalidParam` exception will be raised. If the maximum size of the log is changed, an `AttributeValueChange` event is generated.

Checking

A log's `get_max_size()` method returns its size in bytes.

Log Duration

Overview

In addition to setting fine-grained scheduling intervals for a log to record data, you can also specify a course-grained duration for a log's functionality. By default, a log's functional duration is set to be the log's lifetime. It will start logging records immediately after it is created and continue to log events until it is destroyed.

However, you can program the log to start functioning at a specific time and stop functioning at a later date. Before the log's start time and after its stop time, it will not provide any logging functionality and any schedules set for the log will be invalid. The log will, however, forward events. See "Event Forwarding" on page 199.

Specifying

A log's duration is specified using a `TimeInterval` structure which has the following signature:

```
\\ IDL
struct TimeInterval
{
  TimeT start;
  TimeT stop;
};
```

If you specify a start time of zero, the log will become functional as soon as it is enabled. A stop time of zero causes the log to remain functional until it is destroyed.

Setting

You use a log's `set_interval()` method to set a log's functional interval. It takes a single `TimeInterval` parameter. An `InvalidTimeInterval` is thrown if the start time is before the stop time. If the log's functional duration is successfully changed, an `AttributeValueChange` event is generated.

Note: A race condition could exist when setting the start/stop time. For instance, if a log's start time is too close to the time the `set_interval()` method is invoked, then the time the log may have missed some events that should have been logged before it could be activated.

Record Lifetime

Overview

The lifetime of records in a log determines the amount of time between when the log creates the record and when the log compacts, or deletes the record. By default, all logs have a record life of zero, which specifies that records have an infinite lifespan. However, this also means that the log can not perform any automatic garbage collecting.

For logs with a limited amount of persistent storage space, or for logs that store large volumes of records, you may want to have records expire and be automatically compacted.

Setting

You set a log's record lifetime using the log's `set_record_life()` method. It specifies the record's lifetime in seconds. When you successfully change a log's record lifetime, an `AttributeValueChange` event is generated.

Checking

The `get_max_record_life()` method returns the log's record lifetime setting.

Log QoS Properties

Overview

The telecom log service supports a lightweight QoS framework that specifies the level of assurance that logged records will be stored in a log's persistent data store.

Properties

Log objects support the following QoS settings:

Table 10: *Log QoS settings*

QoS Setting	Log Behavior
QoSNone (default)	Records are buffered in memory when they are logged. The log flushes its memory buffer to the persistent store at intervals specified in the telecom log service's configuration database. This level of service provides no guarantee that logged records will be stored to the persistent store.
QoSFlush	Records are buffered in memory when they are logged. The log's memory buffer is flushed when a client invokes the log's flush() method. This level of service also provides no guarantee that logged records will be written to the persistent store. However, it does provide log clients with greater control over when a log's memory buffer is flushed to the persistent store.
QoSReliability	Records are written directly to the persistent store when they are logged. This level of service guarantees that all records will be available in the persistent store and provides a high level of recoverability in the event of a crash. It will suffer a performance hit due to the increased amount of disk access.

Setting

The set_log_qos() operation sets the quality of service properties of the log. If the QoS properties of a log is changed, an AttributeValueChange event is generated. If set_log_qos() is passed an invalid QoSList type, it will raise a DsLogAdmin::UnsupportedQoS exception.

Flushing the buffer

The `flush()` method writes out a log's memory buffer to the persistent store. It guarantees that all events recorded by the log before the invocation of the `flush()` operation will be written to the persistent store.

Example

sets a log's QoS to `QoSFlush` and then calls `flush()` on it.

Example 49: *Setting a log's QoS properties and flushing its memory buffer*

```java
// Java
//log object obtained previously
QoSType[] qos = new QoSType(1);
qos[0] = QoSFlush.value;

try
  {
  log.set_log_qos(qos);
  }
catch (DsLogAdmin::UnsupportedQoS)
  {
  // handle the exception
  }

// ...

// write the log's memory buffer to disk
log.flush();
```

Availability Status

Overview

The telecom log service updates monitors the availability of all active logs. Depending on scheduling and the amount of data stored in a log, it may not be available for recording new records. Determining a log's availability can provide valuable feedback for clients. For example, a log's clients might generate an alarm if the log is not available because it is full.

States

A log can be in one of three availability states:

Table 11: *Availability states for a log*

State	Log Behavior
On duty	The log is fully functional. It can log new records, forward events, and retrieve records.
Off duty	The log is not scheduled to log new records. All other functionality is still available.
Log full	The log has reached its maximum size and is no longer able to log new records. All other functionality is available.

Checking

The telecom log service provides the `get_availability_status()` method to check a log's availability to log new records. The method returns an `AvailabilityStatus` structure, shown in below.

```
// IDL
struct AvailabilityStatus
{
boolean off_duty;
boolean log_full;
};
```

It is possible that both the `off_duty` and `log_full` fields can be `true` at the same time. A log is on duty if both fields are false and its operational state is `enabled`.

Operational State

Overview

In addition to monitoring logs availability to log new records, the telecom log service also monitors the operation state of log objects. The operation state differs from the availability status of a log in that a log's operational state indicates possible processing errors within a log.

States

Table 12 shows the possible operational states for a log.

Table 12: *Log operational states*

Operational State	Reason
enabled	The log is healthy and its full functionality is available for use.
disabled	The log has encountered a runtime error and is unavailable. The log will not accept any new records and it may not be able to retrieve valid records. The log will still attempt to forward events if its ForwardingState is set to on.

Checking

To check the operational state of a log, invoke its get_operational_state() method. get_operational_state() returns a value of OperationalState, which is an enumerated type with the values enabled and disabled.

State change events

A StateChange event is generated whenever the operational state of a log changes.

Qualities of Service

Overview

In addition to the QoS properties offered by the telecom log service, `NotifyLog` objects can specify notification service level Qualities of Service for events. The additional QoS settings provide greater control over the reliability of messages reaching consumers and the scalability of the telecom log service. The notification service QoS properties include:

- the level of assurance the events will get delivered
- the persistence of client connection information
- an event's priority
- an event's lifetime in the channel
- the order in which the channel discards stale events
- the maximum number of times a proxy tries to contact a client before giving up
- the amount of time between a proxy consumer's calls to `pull()`

For a full listing of the notification service's QoS properties and their descriptions, see "Notification Service Properties" on page 57.

Setting QoS properties

To set notification service level QoS on a `NotifyLog` you use the log's `set_qos()` method. See "Log QoS Properties" on page 216.

Example 50 on page 221 sets a log's `EventReliability` and `ConnectionReliability` QoS to `Persistent`.

Example 50: *Setting notification level QoS on a NotifyLog object*

```
// Java

Property[] qos = new Property(2);
qos[0] = new Property();
qos[0].name = EventReliability.value;
qos[0].value = org.omg.CORBA.ORB.init().create_any();
qos[0].value.insert_short(Persistent.value);
qos[1].name = ConnectionReliability.value;
qos[1].value = org.omg.CORBA.ORB.init().create_any();
qos[1].value.insert_short(Persistent.value);

log.set_qos(qos);
```

Managing the Telecom Log Service

The telecom log service has several configuration variables that determine its behavior. They can control the speed and reliability of the telecom log service.

In this chapter

This chapter discusses the following topics:

Configuring the Telecom Log Service

Overview

The telecom log service can be customized by adjusting the service's configuration settings. Using this mechanism you can set the service's persistence mode, the maximum number or records returned from a query before an iterator object is used("Getting Log Records" on page 180), and the interval between flushes of the log object's internal memory buffer("Flushing the buffer" on page 217).

Configuration scopes

Most of the configuration variables for the telecom log service are found in the following configuration scopes:

- `iona_services.basic_log` - The variables in this scope set the database location, tracing level, persistence mode, and other default settings used by `BasicLog` objects.

- `iona_services.event_log` - The variables in this scope set the database location, tracing level, persistence mode, and other default settings used by `EventLog` objects.

- `iona_services.notify_log` - The variables in this scope set the database location, tracing level, persistence mode, and other default settings used by `NotifyLog` objects.

The initial reference for the telecom log service is set in the configuration's root scope, as are the variables for using the telecom log service with the IONA management service.

Namespaces

The telecom log service's configuration variables are in the following namespaces:

plugins:tlog contains variables to control the general performance of the telecom log service. The variables in this namespace effect all log objects.

plugins:tlog:database contains variables to configure the database used as the persistent store for log objects.

plugins:basic_log contains variables that are related to the generic server plug-in.

plugins:event_log contains variables that are related to the generic server plug-in.

plugins:notify_log contains variables that are related to the generic server plug-in.

In addition to the namespaces that are specifically used to configure telecom log service properties, the following namespace is used to configure the telecom log service's collocated notification service:

plugins:notify contains variables to control the performance of the collocated notification service used by NotifyLog objects. To effect the telecom log service the variables in the plugins:notify namespace must occur in the iona_services.notify_log scope. The variables specified under the iona_services.notification scope do not effect the telecom log service.

Performance tuning variables

Modifying the telecom log service's configuration variables effects the overall performance of the service in terms of the amount of resources it consumes and the speed at which it processes events. You can use the configuration variables to tune the telecom log service's performance to meet you specific needs.

Some of the variables that effect performance are listed in Table 13.

Table 13: *Telecom log service configuration variables*

Variable	Effect
flush_interval	Specifies the time in seconds between automated flushes of a log object's memory buffer. This property only effects log objects with the QoSNone quality of service. Setting the value to 0 disables automatic flushing. The default value is 5 minutes. See "Log QoS Properties" on page 216.
max_records	Specifies the maximum number of records that a query or retrieve operation can return without using an iterator. The default is 100. See "Getting Log Records" on page 180.

Table 13: *Telecom log service configuration variables*

Variable	Effect
`iterator_timeout`	Specifies the lifetime of an inactive iterator object in seconds. Iterator objects that have been inactive for longer than the time specified are reaped. Setting the value to 0 disables iterator reaping. The default value is 4 hours.

Further reading

For a complete listing of the telecom log service's configuration variables and a detailed description of how to set them see the *Application Server Platform Administrator's Guide*.

Running the Telecom Log Service

Starting the service

Like all Orbix services, the telecom log service can be configured to start on demand, to start at system boot, or be started by a script generated by the configuration tool.

You can also manually start the telecom log service with the following command:

Basic Logging

```
C:\Program Files\IONA\asp\6.1\bin\itbasic_log.exe -background
    run  -ORBdomain_name <domain_name> -ORBconfig_domains_dir
    "C:\Program Files\IONA\etc\domains" -ORBname
    iona_services.basic_log
```

Event Logging

```
C:\Program Files\IONA\asp\6.1\bin\itevent_log.exe -background
    run  -ORBdomain_name <domain_name> -ORBconfig_domains_dir
    "C:\Program Files\IONA\etc\domains" -ORBname
    iona_services.event_log
```

Notification Logging

```
C:\Program Files\IONA\asp\6.1\bin\itnotify_log.exe -background
    run  -ORBdomain_name <domain_name> -ORBconfig_domains_dir
    "C:\Program Files\IONA\etc\domains" -ORBname
    iona_services.notify_log
```

Stopping the service

To stop the telecom logging service you can use the stop script generated by the configuration tool or you can use `itadmin`. You stop the telecom log service with the following `itadmin` command:

Basic Logging

```
% basic_log stop
```

Event Logging

```
% event_log stop
```

Notification Logging

```
% notify_log stop
```

Further reading   For a detailed description of using itadmin to start and stop Orbix services
                                    see the *Application Server Platform Administrator's Guide*.

Managing a Deployed Telecom Log Service

Using the telecom log service console

The telecom log service console provides administrators the ability to monitor and control a deployed telecom log service. It provides controls to create and destroy logs, admin objects, proxy objects, and filters. It also provides controls to edit QoS properties, schedules, and lifespans.

To start the telecom log service console use the following command:

```
itlogging_console
```

The console has detailed context sensitive help to guide you in its use.

Part 4

The Java Messaging Service

In this part   This part contains the following chapters:

Developing a JMS Application

The Java Messaging System provides a native messaging solution for all Java applications.

Overview

The Java Messaging System provides facilites for using both point to point messaging or publish / subscribe messaging. Point to point messaging is implemented using *Queues* and publush and subscribe messaging is implemented using *Topics*. Table 14 compares the properties of Queues and Topics.

Table 14: *Queue - Topic feature chart*

	Queue	**Topic**
Most common topologies	one -> one, many -> one	one -> many, many -> many
Action if multiple consumers	Each message goes to only one consumer.	All consumers recieve every message.
Action if no consumers	Messages are retained.	Messages are discarded.
Durable consumers	No, but if a single consumer is being used messages are retained so none will be lost.	Yes
Browse undelivered messages	Yes	No

Table 14: *Queue - Topic feature chart*

	Queue	Topic
Delivery order	FIFO	none

In this chapter   This chapter discusses the following topics:

Using Point to Point Messaging	page 235
Using Publish / Subscribe Messaging	page 249

Using Point to Point Messaging

Overview

In point to point messaging, messages are typically sent between one producer and one consumer. Typically calls to receive messages will block until a message is received; however, senders need not block until a message is received.

JMS *Queues* are used to implement point to point messaging between clients. A Queue delivers a message once and only once. Typically, a Queue will only have one client consuming messages although JMS does not prohibit connecting more than one consumer. A JMS Queue also provides for asynchronous point to point messaging.

As the name implies, a Queue enforces a FIFO order of message delivery. Messages are placed at the end of the queue when they are posted and cannot be consumed out of order. Once a message has been consumed, it is popped off the queue and cannot be consumed again unless it is redelivered. A consumer can browse a queue to see what messages are waiting to be consumed, and can acknowledge that messages have been received.

In this section

This section discussed the following topics:

Creating a Queue

Overview

Queues are considered administered objects in JMS and are maintained by the service. Applications using JMS Queues need to use a JNDI lookup to get a reference to an existing Queue.

By default the IONA JMS initializes two default Queues called `"iona:jms/queue/queue0"` and `"iona:jms/queue.queue1"`. `queue0` is used by the included demo programs; `queue1` is left free.

> **Note:** Because Queues only deliver a message to one consumer, it is advisable to ensure that each application uses a unique Queue.

New Queues can be created in one of two ways:

- Using the management console
- Programatically

Using the management console

The IONA management service provides a `createQueue` operation to create new Queues. For more information on using the management service with JMS see "Managing JMS with the Management Service" on page 270.

Programatically

To create a Queue programatically, five steps are required:

1. Create an initial context.

2. Get a reference to the queue connection factory using a JNDI lookup for `"iona:jms/queue/connectionFactory"` as shown in Example 51.

Example 51: *Looking up the queue connection factory.*

```
// Java
import javax.jms.Queue;
import javax.jms.QueueConnectionFactory;

// Context ctx obtained previously
QueueConnectionFactory qconFactory = (QueueConnectionFactory)
    ctx.lookup("iona:jms/queue/connectionFactory");
```

3. Create a connection using the connection factory as shown in Example 52.

Example 52: *Create a connection.*

```java
// Java
import com.iona.jms.api.ITQueueConnection;

// Connection factory qconFactory obtained in previous example
ITQueueConnection queueConn = (ITQueueConnection)
   qconFactory.createQueueConnection();
```

4. Get a reference to the `DestinationAdmin` from the newly created connection as shown in Example 53.

Example 53: *Obtaining a DestinationAdmin*

```java
// Java
import com.iona.jms.api.admin.ITDestinationAdmin;

// Connection queueConn obtained in previous example
ITDestinationAdmin destAdmin = queueConn.getDestinationAdmin();
```

5. Create a Queue using the `DestinationAdmin`'s `createDestination()` operation as shown in Example 54.

Example 54: *Creating a Queue*

```java
// Java
import javax.jms.Queue;

// DestinationAdmin destAdmin obtained in previous example
Queue queue = (Queue) destAdmin.createDesination(queue_name,
   ITDestination.QUEUE, null);
```

`createDestination()` takes the follwoing parameters:

queue_name	The name of the new queue.
type	The type of destination to create. `ITDestination.QUEUE` is specified because the new destination is a Queue.
properties	The Java properties for the new destination.

Implementing a Point to Point Message Producer

Overview

A client wishing to send messages using a JMS Queue must perform the following tasks:

1. Get a reference to a Queue and the queue connection factory.
2. Create a `QueueConnection` using the connection factory.
3. Create a `QueueSession` using the connection.
4. Create a `QueueSender` using the session.
5. Create and send messages.
6. Close the connection.

Getting the administered objects

The queue connection factory and all existing Queues are administered objects and must be discovered using the JNDI lookup.

The reference for the queue connection factory is `"iona:jms/queue/connectionFactory"`. The reference for a Queue takes the form `"iona:jms/queue/`*queueName*`"` where *queueName* specifies the name of the Queue.

> **Note:** A Queue must exist before it can be used by a client program. For information on creating a Queue, see "Creating a Queue" on page 236

Example 55 shows how to obtain references for Queue `sportsQueue` and the queue connection factory.

Example 55: *Obtaining references to the administered objects*

```
// Java
import javax.jms.Queue;
import javax.jms.QueueConnectionFactory;

// InitialContect ctx obtained previously
// Lookup the queue connection factory
QueueConnectionfactory qconnFactory = (QueueConnectionFactory)
   ctx.lookup("iona:jms/queue/connectionFactory");

// Lookup the Queue sportsQueue
Queue queue = (Queue) ctx.lookup("iona:jms/queue/sportsQueue");
```

Creating a connection

A JMS `QueueConnection` maintains the active connection information between the client and JMS. It provides operations for stopping and starting the flow of messages. However, a message producer would be unlikely to use these operations because their behavior only effect the ability of message consumers to receive messages from a connection. A connection that has been stopped will still accept messages from a message producer and will deliver the messages once the connection is restarted.

The `QueueConnection` also provides the operation for creating `Session` objects which manage the actual production and consumption of messages.

`QueueConnection`s are created using the queue connection factory's `createQueueConnection()` operation which takes no arguments. Example 56 shows how to create a `QueueConenction`.

Example 56: *Creating a QueueConnection*

```java
// Java
import javax.jms.QueueConnection;

// Connection factory qconFactory obtained previously
QueueConnection queueConn = (QueueConnection)
   qconFactory.createQueueConnection();
```

Creating a session

A `QueueSession` is created using the `createQueueSession()` operation of a `QueueConnection`. This operation has the following signature:

```
QueueSession createQueueSession(Boolean transacted, int
   acknowledge);
```

`createQueueSession()` takes two parameters.

transacted   Specifies if the `Session` is transacted. Can be either `true` or `false`.

acknowledge	Specifies how the receipt of messages will be acknowledged by the Session. Can be take one of three values:
	AUTO_ACKNOWLEDGE specifies that the session automatically acknowledges the receipt of all messages immediately upon the receiving client's return from the recieve() operation.
	CLIENT_ACKNOWLEDGE specifies that message receivers must acknowledge the receipt of a message by calling the message's acknowledge() method.
	DUPS_OK_ACKNOWLEDGE specifies that the session can lazily acknowledge the delivery of messages. This may result in duplicate messages being delivered to clients.

A QueueSession provides a single-threaded context for sending and receiving messages. It provides operations for creating QueueSenders, QueueRecievers, QueueBrowsers, and messages.

The code in Example 57 creates a non-transacted QueueSession that uses lazy acknowledgement of messages.

Example 57: *Creating a QueueSession*

```Java
// Java
import javax.jms.Session;
import javax.jms.QueueSession;

// Connection queueConn obtained previously
QueueSession queueSession = queueConn.createQueueSession(false,
    Session.DUPS_OK_ACKNOWLEDGE);
```

Creating a message sender

In point to point messaging, messages are sent using a QueueSender object. A QueueSender is created from a QueueSession using the session's createSender() operation. The operation has the following signature:

```
QueueSender createSender(Queue queue);
```

`createSender()` takes a single parameter which is the `Queue` to which the sender is going to send messages.

A `QueueSender` is also responsible for specifying the level of guarantee, or delivery mode, that is used when sending messages. This is done using the `QueueSender`'s `setDeliveryMode()` operation. This operation takes a single parameter which can take one of the following values:

DeliveryMode.NON_PERSISTENT specifies that there is no guarantee that a message will be delivered if the JMS Broker or other component of JMS fails while the message is in transit.

DeliveryMode.PERSISTENT specifies that the message is guaranteed to be delivered even if the JMS Broker or other component of JMS fails while the message is in transit.

The code in Example 58 creates a `QueueSender` and sets its delivery mode to `PERSISTENT`.

Example 58: *Creating a persistent QueueSender*

```java
// Java
import javax.jms.QueueSender;
import javax.jms.DeliveryMode;

// Session queueSession obtained in a previous example
// Queue queue obtained in a previous example
QueueSender qSender = queueSession.createSender(queue);
qSender.setDeliveryMode(DeliveryMode.PERSISTENT);
```

Creating and sending a message

JMS messages come in several different flavors depending on the type of information you wish to use as message. The messages are created from a `Session` using one of the session's create message operations.

Messages are sent using the `QueueSender`'s `send()` operation.

241

The code in Example 59 creates a simple text message containing the string `"pulchritudinous"` and sends it.

Example 59: *Sending a simple text message*

```java
// Java
import javax.jms.TextMessage;

// Session queueSession and QueueSender qSender obtained
// in a previous example
TextMessage message =
    quueueSession.createTextMessage("pulchritudinous");
qSender.send(message);
```

Closing the connection

A `QueueConnection` is resource intensive and should be closed one it is no longer in use. Also, because it is possible for a number of objects to hold references to the connection, the JVM's garbage collection routine will not recover the resources in a timely or reliable manner.

Connections are closed using the `close()` operation. Once the call to `close()` is made, all `Session`s, and their associated `QueueSender`s and messages, are destroyed.

Example 60 shows how to close a `QueueConnection`.

Example 60: *Closing a QueueConnection*

```java
// Java
// Connection queueConn obtained in a previous example
queueConn.close();
```

Implementing a Point to Point Message Consumer

Overview

A client wishing to receive messages using a JMS Queue must perform the following tasks:

1. Get a reference to a Queue and the queue connection factory.

2. Create a `QueueConnection` using the connection factory.

3. Create a `QueueSession` using the connection.

4. Create a `QueueReciever` using the session.

5. Start the `QueueConnetion` to begin the flow of messages.

6. Receive and process messages.

7. Close the connection.

Getting the administered objects

The queue connection factory and all existing Queues are administered objects and must be discovered using the JNDI lookup.

The reference for the queue connection factory is `"iona:jms/queue/connectionFactory"`. The reference for a Queue takes the form `"iona:jms/queue/`*queueName*`"` where *queueName* specifies the name of the Queue.

> **Note:** A Queue must exist before it can be used by a client program. For information on creating a Queue, see "Creating a Queue" on page 236

Example 61 shows how to obtain references for Queue `sportsQueue` and the queue connection factory.

Example 61: *Obtaining references to the administered objects*

```java
// Java
import javax.jms.Queue;
import javax.jms.QueueConnectionFactory;

// InitialContect ctx obtained previously
// Lookup the queue connection factory
QueueConnectionfactory qconnFactory = (QueueConnectionFactory)
    ctx.lookup("iona:jms/queue/connectionFactory");

// Lookup the Queue sportsQueue
Queue queue = (Queue) ctx.lookup("iona:jms/queue/sportsQueue");
```

Creating a connection

A JMS `QueueConnection` maintains the active connection information between the client and JMS. It provides operations for stopping and starting the flow of messages. When a connection is stopped, using the `stop()` operation, message consumers will not receive any messages. Also, message time-out values will continue to be enforced and therefore messages may time-out while the connection is stopped. Once the connection is restarted, using the `start()` operation, message consumers will again begin receiving messages.

Note: Message producers can continue to send messages while the connection is stopped. These new messages will be delivered when the connection is restarted.

The `QueueConnection` also provides the operation for creating `Session` objects which manage the actual production and consumption of messages.

QueueConnections are created using the queue connection factory's
createQueueConnection() operation which takes no arguments.
Example 62 shows how to create a QueueConenction.

Example 62: *Creating a QueueConnection*

```
// Java
import javax.jms.QueueConnection;

// Connection factory qconFactory obtained previously
QueueConnection queueConn = (QueueConnection)
   qconFactory.createQueueConnection();
```

When a QueueConneciton is first created, it is stopped and message
receivers cannot receive messages until it is explicitly started. While the
connection can be started immediately after it is created, this could result in
messages being delivered before the message receiver is ready to process
them. It is best to start the connection after the message receiver is
initialized and ready to process messages.

Creating a session

A QueueSession is created using the createQueueSession() operation of a
QueueConnection. This operation has the following signature:

```
QueueSession createQueueSession(Boolean transacted, int
   acknowledge);
```

createQueueSession() takes two parameters.

transacted   Specifies if the Session is transacted. Can be either true
or false.

| acknowledge | Specifies how the receipt of messages will be acknowledged by the Session. Can be take one of three values: |
| | |

AUTO_ACKNOWLEDGE specifies that the session automatically acknowledges the receipt of all messages immediately upon the receiving client's return from the recieve() operation.

CLIENT_ACKNOWLEDGE specifies that message receivers must acknowledge the receipt of a message by calling the message's acknowledge() method.

DUPS_OK_ACKNOWLEDGE specifies that the session can lazily acknowledge the delivery of messages. This may result in duplicate messages being delivered to clients.

A QueueSession provides a single-threaded context for sending and receiving messages. It provides operations for creating QueueSenders, QueueRecievers, QueueBrowsers, and messages.

The code in Example 63 creates a non-transacted QueueSession that uses lazy acknowledgement of messages.

Example 63: *Creating a QueueSession*

```Java
// Java
import javax.jms.Session;
import javax.jms.QueueSession;

// Connection queueConn obtained previously
QueueSession queueSession = queueConn.createQueueSession(false,
    Session.DUPS_OK_ACKNOWLEDGE);
```

Creating a message receiver

In point to point messaging, messages are received by a QueueReciever object. A QueueReciever is created from a QueueSession using the session's createReciever() operation. The operation has the following signature:

```
QueueReciever createReciever(Queue queue);
```

createReciever() takes a single parameter which is the Queue from which the receiver is going to recieve messages.

The code in Example 64 creates a QueueReciever.

Example 64: *Creating a QueueReciever*

```Java
// Java
import javax.jms.QueueReciever;

// Session queueSession obtained in a previous example
// Queue queue obtained in a previous example
QueueReciever qReciever = queueSession.createReciever(queue);
```

Starting the connection and receiving messages

Once the QueueReciever is created and any other initialization required for message processing is completed, the connection needs to be started to begin the flow of messages. The connection is started using the start() operation on the QueueConnection.

After starting the connection, the QueueReciever can begin synchronously receiving messages using its recieve() operation. recieve() blocks until a message is consumed from the queue. The operation can take a parameter that specifies the amount of time, in milliseconds, to block before timing out.

If a client wishes to receive messages asynchronously, it can register a MessageListener, which will notify the client when messages are ready..

Example 65 shows the code to start a connection and receive a text message. The QueueReciever only blocks for 4 seconds before timing out.

Example 65: *Receiving a text message*

```Java
// Java
import javax.jms.TextMessage;

// Connection queueConn and QueueReciever qReciever obtianed
// in a previous example
queueConn.start();
TextMessage message = qReciever.recieve(4000);
if (message != null)
 System.out.println("Message recieved: " + message.getText());
```

If a message is received from the queue, the message is printed out using the `getText()` operation defined on a JMS `TextMessage`. If no message is received from the queue, `recieve()` returns a `null`.

Closing the connection

A `QueueConnection` is resource intensive and should be closed one it is no longer in use. Also, because it is possible for a number of objects to hold references to the connection, the JVM's garbage collection routine will not recover the resources in a timely or reliable manner.

Connections are closed using the `close()` operation. Once the call to `close()` is made, all `Session`s, and their associated `QueueSender`s and messages, are destroyed.

Example 66 shows how to close a `QueueConnection`.

Example 66: *Closing a QueueConnection*

```
// Java
// Connection queueConn obtained in a previous example
queueConn.close();
```

Using Publish / Subscribe Messaging

Overview

In publish / subscribe messaging, many message producers forward messages to many message consumers. This style of messaging is typically asynchronous, meaning that call to receive messages do not block. The message consumers continue to process information while waiting for new messages to arrive.

JMS *Topics* are used to implement publish / subscribe messaging between JMS clients. Topics allow anonymous publishers, *TopicPublishers*, and anonymous suppliers, *TopicSubscriber*, to connect and disconnect at random intervals. The Topic ensures that all messages are delivered to each connected consumer at least once. If a consumer wishes to receive messages even when it is disconnected from the Topic it must register using a durable subscription.

Topics also provide a means to ensure that messages are delivered to all connected consumers reliably. Messages published as `NON_PERSISTENT` are not guaranteed to arrive to all connected consumers. Messages published `PERSISTENT` are guaranteed to arrive to each connected consumer at most once.

In this section

This sections discusses the following topics:

Creating a Topic

Overview

Topics are considered administered objects in JMS and are maintained by the service. Applications using JMS Topics need to use a JNDI lookup to get a reference to an existing Topic.

By default the IONA JMS initializes two default Queues called `"iona:jms/queue/topic0"` and `"iona:jms/queue.topic1"`. `topic0` is used by the included demo programs; `topic1` is left free.

Topics can be created in one of two ways:

- Using the management console
- Programatically

Using the management console

The IONA management service provides a `createTopic` operation to create new Topics. For more information on using the management service with JMS see "Managing JMS with the Management Service" on page 270.

Programatically

To create a Topic programatically, five steps are required:

1. Create an initial context.
2. Get a reference to the topic connection factory using a JNDI lookup for `"iona:jms/topic/connectionFactory"` as shown in Example 67.

Example 67: *Looking up the topic connection factory.*

```
// Java
import javax.jms.TopicConnectionFactory;

// Context ctx obtained previously
TopicConnectionFactory tconFactory = (TopicConnectionFactory)
    ctx.lookup("iona:jms/topic/connectionFactory");
```

3. Create a connection using the connection factory as shown in
   Example 68.

Example 68: *Create a connection.*

```java
// Java
import com.iona.jms.api.ITTopicConnection;

// Connection factory tconFactory obtained in previous example
ITTopicConnection topicConn = (ITTopicConnection)
   tconFactory.createTopicConnection();
```

4. Get a reference to the DestinationAdmin from the newly created
   connection as shown in Example 69.

Example 69: *Obtaining a DestinationAdmin*

```java
// Java
import com.iona.jms.api.admin.ITDestinationAdmin;

// Connection topicConn obtained in previous example
ITDestinationAdmin destAdmin = topicConn.getDestinationAdmin();
```

5. Create a Topic using the DestinationAdmin's createDestination()
   operation as shown in Example 70.

Example 70: *Creating a Topic*

```java
// Java
import javax.jms.Topic;

// DestinationAdmin destAdmin obtained in previous example
Topic topic = (Topic) destAdmin.createDesination(topic_name,
   ITDestination.TOPIC, null);
```

createDestination() takes the follwoing parameters:

topic_name   The name of the new topic.

type   The type of destination to create. ITDestination.TOPIC
              is specified because the new destination is a Topic.

properties   The Java properties for the new destination.

Implementing a Message Publisher

Overview

A client wishing to publish messages to a JMS Topic must perform the following tasks:

1. Get a reference to a Topic and the topic connection factory.
2. Create a `TopicConnection` using the connection factory.
3. Create a `TopicSession` using the connection.
4. Create a `TopicPublisher` using the session.
5. Create and publish messages.
6. Close the connection.

Getting the administered objects

The queue connection factory and all existing Topics are administered objects and must be discovered using the JNDI lookup.

The reference for the topic connection factory is `"iona:jms/topic/connectionFactory"`. The reference for a Queue takes the form `"iona:jms/topic/`*topicName*`"` where *topicName* specifies the name of the Queue.

> **Note:** A Topic must exist before it can be used by a client program. For information on creating a Topic, see "Creating a Topic" on page 250

Example 71 shows how to obtain references for Topic `sportsTopic` and the topic connection factory.

Example 71: *Obtaining references to the administered objects*

```
// Java
import javax.jms.Topic;
import javax.jms.TopicConnectionFactory;

// InitialContect ctx obtained previously
// Lookup the topic connection factory
TopicConnectionfactory tconnFactory = (TopicConnectionFactory)
   ctx.lookup("iona:jms/topic/connectionFactory");

// Lookup the Topic sportsTopic
Topic topic = (Topic) ctx.lookup("iona:jms/topic/sportsTopic");
```

Creating a connection

A JMS `TopicConnection` maintains the active connection information between the client and JMS. It provides operations for stopping and starting the flow of messages. However, the state of the connection does not effect a publisher's ability to publish message to the Topic. Therefore message publishers would most likely not change the state of the Connection.

The `TopicConnection` also provides the operation for creating `Session` objects which manage the actual production and consumption of messages.

`TopicConnection`s are created using the topic connection factory's `createTopicConnection()` operation which takes no arguments. Example 72 shows how to create a `TopicConenction`.

Example 72: *Creating a TopicConnection*

```
// Java
import javax.jms.TopicConnection;

// Connection factory tconFactory obtained previously
TopicConnection topicConn = (TopicConnection)
   tconFactory.createTopicConnection();
```

Creating a session

A `TopicSession` is created using the `createTopicSession()` operation of a `TopicConnection`. This operation has the following signature:

```
TopicSession createTopicSession(Boolean transacted, int
   acknowledge);
```

`createQueueSession()` takes two parameters.

transacted   Specifies if the `Session` is transacted. Can be either `true` or `false`.

253

acknowledge
Specifies how the receipt of messages will be acknowledged by the `Session`. Can be take one of three values:

AUTO_ACKNOWLEDGE specifies that the session automatically acknowledges the receipt of all messages immediately upon the receiving client's return from the `recieve()` operation.

CLIENT_ACKNOWLEDGE specifies that message receivers must acknowledge the receipt of a message by calling the message's `acknowledge()` method.

DUPS_OK_ACKNOWLEDGE specifies that the session can lazily acknowledge the delivery of messages. This may result in duplicate messages being delivered to clients.

A `TopicSession` provides a single-threaded context for sending and receiving messages. It provides operations for creating `TopicPublisher`s, `TopicSubscriber`s, and messages.

The code in Example 73 creates a non-transacted `TopicSession` that uses client acknowledgement of messages.

Example 73: *Creating a TopicSession*

```
// Java
import javax.jms.Session;
import javax.jms.TopicSession;

// Connection TopicConn obtained previously
TopicSession topicSession = topicConn.createTopicSession(false,
    Session.CLIENT_ACKNOWLEDGE);
```

Creating a message publisher

In publish / subscribe messaging, messages are published using a `TopicPublisher` object. A `TopicPublisher` is created from a `TopicSession` using the session's `createPublisher()` operation. The operation has the following signature:

```
TopicPublisher createPublisher(Topic topic);
```

createPublisher() takes a single parameter which is the Topic to which the publisher publishes messages.

A TopicPublisher is also responsible for specifying the level of guarantee, or delivery mode, that is used when sending messages. This is done using the TopicPublisher's setDeliveryMode() operation. This operation takes a single parameter which can take one of the following values:

DeliveryMode.NON_PERSISTENT specifies that there is no guarantee that a message will be delivered if the JMS Broker or other component of JMS fails while the message is in transit.

DeliveryMode.PERSISTENT specifies that the message is guaranteed to be delivered even if the JMS Broker or other component of JMS fails while the message is in transit.

The code in Example 74 creates a TopicPublisher and sets its delivery mode to PERSISTENT.

Example 74: *Creating a persistent TopicPublisher*

```Java
// Java
import javax.jms.TopicPublisher;
import javax.jms.DeliveryMode;

// Session topicSession obtained in a previous example
// Topic topic obtained in a previous example
TopicPublisher tPub = topicSession.createPublisher(topic);
tPub.setDeliveryMode(DeliveryMode.PERSISTENT);
```

Creating and publishing messages

JMS messages come in several different flavors depending on the type of information you wish to use as message. The messages are created from a Session using one of the session's create message operations.

Messages are sent using the TopicPublisher's publish() operation.

The code in Example 75 creates a simple text message containing the string
`"pulchritudinous"` and publishes it.

Example 75: *Publishing a simple text message*

```
// Java
import javax.jms.TextMessage;

// Session topicSession and TopicPublisher tPub obtained
// in a previous example
TextMessage message =
    topicSession.createTextMessage("pulchritudinous");
tPub.publish(message);
```

Closing the connection

A `TopicConnection` is resource intensive and should be closed one it is no
longer in use. Also, because it is possible for a number of objects to hold
references to the connection, the JVM's garbage collection routine will not
recover the resources in a timely or reliable manner.

Connections are closed using the `close()` operation. Once the call to
`close()` is made, all `Session`s, and their associated `TopicPublisher`s and
messages, are destroyed.

Example 76 shows how to close a `TopicConnection`.

Example 76: *Closing a TopicConnection*

```
// Java
// Connection topicConn obtained in a previous example
topicConn.close();
```

Implementing a Subscriber

Overview

A client wishing to publish messages to a JMS Topic must perform the following tasks:

1. Get a reference to a Topic and the topic connection factory.

2. Create a `TopicConnection` using the connection factory.

3. Create a `TopicSession` using the connection.

4. Create a `TopicSubscriber` using the session.

5. Create and publish messages.

6. Close the connection.

Getting the administered objects

The queue connection factory and all existing Topics are administered objects and must be discovered using the JNDI lookup.

The reference for the topic connection factory is `"iona:jms/topic/connectionFactory"`. The reference for a Queue takes the form `"iona:jms/topic/`*topicName*`"` where *topicName* specifies the name of the Queue.

> **Note:** A Topic must exist before it can be used by a client program. For information on creating a Topic, see "Creating a Topic" on page 250

Example 77 shows how to obtain references for Topic `sportsTopic` and the topic connection factory.

Example 77: *Obtaining references to the administered objects*

```java
// Java
import javax.jms.Topic;
import javax.jms.TopicConnectionFactory;

// InitialContect ctx obtained previously
// Lookup the topic connection factory
TopicConnectionfactory tconnFactory = (TopicConnectionFactory)
    ctx.lookup("iona:jms/topic/connectionFactory");

// Lookup the Topic sportsTopic
Topic topic = (Topic) ctx.lookup("iona:jms/topic/sportsTopic");
```

Creating a connection

A JMS `TopicConnection` maintains the active connection information between the client and JMS. It provides operations for stopping and starting the flow of messages. When a connection is stopped, using the `stop()` operation, message consumers will not receive any messages. Also, message time-out values will continue to be enforced and therefore messages may time-out while the connection is stopped. Once the connection is restarted, using the `start()` operation, message consumers will again begin receiving messages.

Note: Message producers can continue to send messages while the connection is stopped. These new messages will be delivered when the connection is restarted.

The `TopicConnection` also provides the operation for creating `Session` objects which manage the actual production and consumption of messages.

`TopicConnection`s are created using the topic connection factory's `createTopicConnection()` operation which takes no arguments. Example 78 shows how to create a `TopicConenction`.

Example 78: *Creating a TopicConnection*

```Java
// Java
import javax.jms.TopicConnection;

// Connection factory tconFactory obtained previously
TopicConnection topicConn = (TopicConnection)
   tconFactory.createTopicConnection();
```

When a `TopicConneciton` is first created, it is stopped and subscribers cannot receive messages until it is explicitly started. While the connection can be started immediately after it is created, this could result in messages being delivered before the subscriber is ready to process them. It is best to start the connection after the subscriber is initialized and ready to process messages.

Creating a session

A `TopicSession` is created using the `createTopicSession()` operation of a `TopicConnection`. This operation has the following signature:

```
TopicSession createTopicSession(Boolean transacted, int
   acknowledge);
```

`createQueueSession()` takes two parameters.

`transacted`	Specifies if the `Session` is transacted. Can be either `true` or `false`.
`acknowledge`	Specifies how the receipt of messages will be acknowledged by the `Session`. Can be take one of three values:

AUTO_ACKNOWLEDGE specifies that the session automatically acknowledges the receipt of all messages immediately upon the receiving client's return from the `recieve()` operation.

CLIENT_ACKNOWLEDGE specifies that message receivers must acknowledge the receipt of a message by calling the message's `acknowledge()` method.

DUPS_OK_ACKNOWLEDGE specifies that the session can lazily acknowledge the delivery of messages. This may result in duplicate messages being delivered to clients.

A `TopicSession` provides a single-threaded context for sending and receiving messages. It provides operations for creating `TopicPublisher` objects, `TopicSubscriber` objects, and messages.

The code in Example 79 creates a non-transacted `TopicSession` that uses client acknowledgement of messages.

Example 79: *Creating a TopicSession*

```java
// Java
import javax.jms.Session;
import javax.jms.TopicSession;

// Connection TopicConn obtained previously
TopicSession topicSession = topicConn.createTopicSession(false,
    Session.CLIENT_ACKNOWLEDGE);
```

Creating a message subscriber

In publish / subscribe messaging messages are consumed by a `TopicSubscriber` object. A TopicSubscriber is created from a `TopicSession` using the session's `createSubscriber()` operation. The operation has the following signature:

```
TopicSubscriber createSubscriber(Topic topic);
```

`createSubscriber()` takes a single parameter which is the `Topic` from which the subscriber consumes messages.

The code in Example 80 creates a `TopicSubscriber`.

Example 80: *Creating a TopicSubscriber*

```java
// Java
import javax.jms.TopicSubscriber;

// Session topicSession obtained in a previous example
// Topic topic obtained in a previous example
TopicSubscriber tSub = topicSession.createSubscriber(topic);
```

Starting the connection and consuming messages

Once the `TopicSubscriber` is created and any other initialization required for message processing is completed, the connection needs to be started to begin the flow of messages. The connection is started using the `start()` operation on the `TopicConnection`.

After starting the connection, the `TopicSubscriber` can begin synchronously consuming messages using its `recieve()` operation. `recieve()` blocks until a message is consumed from the topic. The operation can take a parameter that specifies the amount of time, in milliseconds, to block before timing out.

If a client wishes to consume messages asynchronously, it can register a `MessageListener`, which will notify the client when messages are ready. .

Example 81 shows the code to start a connection and consume a text message. The TopicSubscriber blocks for 4 seconds before timing out.

Example 81: *Consuming a text message*

```
// Java
import javax.jms.TextMessage;

// Connection topicConn and TopicSubscriber tSub obtianed
// in a previous example
topicConn.start();
TextMessage message = (TextMessage) tSub.recieve(4000);
if (message != null)
 System.out.println("Message recieved: " + message.getText());
```

If a message is consumed from the topic, the message is printed out using the getText() operation defined on a JMS TextMessage. If no message is consumed from the topic, recieve() returns a null.

Closing the connection

A TopicConnection is resource intensive and should be closed one it is no longer in use. Also, because it is possible for a number of objects to hold references to the connection, the JVM's garbage collection routine will not recover the resources in a timely or reliable manner.

Connections are closed using the close() operation. Once the call to close() is made, all Session objects, and their associated QueueSender objects and messages, are destroyed.

Example 82 shows how to close a TopicConnection.

Example 82: *Closing a TopicConnection*

```
// Java
// Connection topicConn obtained in a previous example
topicConn.close();
```

Managing JMS

The Java Messaging Service is fully configurable to handle a variety of deployment scenarios.

In this chapter

This chapter discusses the following topics:

JMS Configuration

Overview

To maximize the service's scalability and functionality, JMS is highly configurable. Like other Orbix services, JMS has a number of user editable configuration variables that control its behavior. For instance, you can control the maximum amount of active JDBC connections the broker can offer and what database JMS uses for a persistent store.

Configuration scope

The JMS broker's configuration variables are in the `iona_services.jms` configuration scope. JMS client configuration information, such as the initial references to the JMS broker, are found in the global scope, or may be scoped by the client's ORB name.

Initial References

JMS clients require two initial references:

IT_JMSMessageBroker provides the initial reference to the JMS message broker.

IT_JMSServerContext is used to support JNDI lookup of JMS destinations and connection factories. See "JNDI" on page 265.

Namespaces

Other than the initial reference variables, the JMS configuration variables only affect the behavior of the JMS broker. They are contained in several namespaces under the JMS configuration scope. The namespaces are:

persistence:jdbc controls what implementation is being used for the service's persistent store. Orbix currently supports Cloudscape, Oracle, and a purely in-memory implementation.

destinations controls the default queue and topic names that the service creates at start up.

plugins:jms specifies the service's type of persistence and its well known address.

thread_pool specifies the behavior of the services thread pool.

factory specifies a username and password for accessing the
`javax.jms.ConnectionFactory` object.

Other variables

JMS also has variables that specify a replica name, if the service's management features are active, and the number of threads available to the service. For a complete listing of the configuration variables see the *Orbix Configuration Reference Guide*.

JNDI

JMS clients use JNDI to get references to connection factories and message destinations. For JMS a URL-based naming scheme is used for identifying JMS objects. The JNDI entries for JMS are:

```
QueueConnectionFactory: "iona:jms/queue/connectionFactory"
TopicConnectionFactory: "iona:jms/topic/connectionFactory"
Queue                 : "iona:jms/queue/queue_name"
Topic                 : "iona:jms/topic/topic_name"
```

To enable the URL naming scheme, you must set the JNDI `java.naming.factory.url.pkgs` property to `com.iona.jms.naming`. You can do this using system properties, a properties file, or programmatically. Example 83 shows how to set the property programmatically.

Example 83: *Setting the JNDI naming package to use JMS URL based names*

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.jms.Topic;

// ...

Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "com.iona.jms.naming");
InitialContext ctx = new InitialContext(env);
Topic topic = (Topic)ctx.lookup("iona:jms/topic/topic0");
```

Running JMS

Overview   JMS has several start-up configurations depending upon how it is deployed.

In this section   This section discusses the following topics:

Starting the JMS Broker

Overview

JMS is configured to start in the same manner as all other Orbix services. In addition, the JMS broker can be started in one of four ways:

- Pure-Java standalone mode
- Native standalone mode
- NT service
- Embedded mode

Pure-Java standalone mode

To start the JMS broker in pure-Java standalone mode, you run it via the JRE.

```
java -Dcloudscape.system.home=db_dir\var\domain_name\dbs\jms
   -DORBdomain_name=domain_name
   -DORBconfig_domains_dir=domain_dir
   com.iona.jms.server.JMSBroker
```

db_dir	The directory where Orbix log and database files are stored. You specified this location when you configured the domain.
domain_name	The name of your configuration domain.
domain_dir	The directory where Orbix configuration files are stored. You specified this location when you configured the domain.

The cloudscape.system.home system property tells Cloudscape where the JMS database resides. If this property is not set, Cloudscape assumes the database should be written in the current working directory.

Note: This system property only needs to be set if you are using Cloudscape as the JMS persistent store implementation. Oracle may have other system properties that need to be set.

Native standalone mode   The JMS broker can also be run under the Orbix's native JVM wrapper.

To start the JMS broker in native standalone mode, use the following command:

```
itjms
```

NT service   The JMS broker, like all other Orbix services, can be configured to start as an NT service on systems running Windows. When running as an NT service, the JMS broker runs in native standalone mode. For more information, see the *Application Server Platform Administrator's Guide*.

Embedded mode   The JMS broker can also be started from within a Java application that wishes to use it. Once running, the JMS broker will be available for any Java client that wishes to use it.

To start the JMS broker from within a Java application use the following code:

```
import com.iona.jms.api.ITMessageService;

//...

ITMessageService msgSvc = ITMessageService.init(args);
msgSvc.start(true);
```

Shutting Down the JMS Broker

Overview

The JMS broker may be stopped in one of three ways:

- Using the JRE
- Using itadmin
- Programmatically

Using the JRE

To shut down the JMS broker using the JRE, use the following command:

```
java -Dcloudscape.system.home=db_dir\var\domain_name\dbs\jms
    -DORBdomain_name=domain_name
    -DORBconfig_domains_dir=domain_dir
    com.iona.jms.server.JMSBroker shutdown
```

Using itadmin

To shut down the JMS broker using itadmin, use the following command:

```
itadmin jms stop
```

Programmatically

If the JMS broker was started by a Java application, the application can shut it down using the shutdown() operation of the ITMessageService used to start the broker.

```
// msgSvc obtained in previous code sample
msgSvc.shutdown();
```

Managing JMS with the Management Service

Overview

JMS includes instrumentation for the IONA management service and can be managed through the IONA Administrator Console. Two MBean types are exposed by JMS. One for the JMS broker and one for each JMS destination.

Configuring JMS to be managed

To enable JMS management using the IONA management service and IONA Administrator Console there are two configuration variables that must be set:

```
plugins:jms:is_managed = "true";
instrumentation:enabled = "true";
```

JMS can also be managed through a light-weight JMX wed adaptor that is provided with the product. To enable this form of management set the following configuration variables:

```
jmx:adaptor:enabled = "true";
jmx:adaptor:port = "port_number";
```

Broker Administration

If you have enabled JMS management in your configuration, the JMS broker's MBean is registered as the root MBean for the service. It exposes the following attributes:

Table 15: *JMS broker MBean attributes*

Attribute	Description
InstrumentationEnabled	A boolean determining if verbose statistics are being generated for JMS.
InstrumentationStartTime	The time at which verbose statistic generation began.
MessageThroughput	A read-only attribute showing the number of JMS messages processed since the `InstrumentationStartTime`.
DataThroughput	A read-only attribute showing the amount of JMS message data processes since the `InstrumentationStartTime`.

Table 15: *JMS broker MBean attributes*

Attribute	Description
MessageInProgress	A read-only attribute showing the number of JMS messages currently being processed.
DataInProgress	A read-only attribute showing the amount of JMS message data currently being processed.
ActiveQueues	A read-only list of `ObjectNames` representing the currently active queues.
ActiveTopics	A read-only list of `ObjectNames` representing the currently active topics.

The JMS broker's MBean also exposes the following operations:

Table 16: *JMS broker MBean operations*

Operation	Description
createQueue	Creates a new named queue.
removeQueue	Removes an existing queue.
createTopic	Creates a new named topic.
removeTopic	Removes an existing topic.
shoutdownBroker	Shuts down the JMS broker.

Destination Administration

If you have enabled JMS management in your configuration, each destination used by the service will have an MBean registered with the management service. The destination MBeans expose the following attributes:

Table 17: *JMS destination MBean attributes*

Attribute	Description
URL	The URL used to lookup this destination in the JNDI.
MessageThroughput	A read-only attribute showing the throughput of the destination in terms of JMS messages since the `InstrumentationStartTime`.

Table 17: *JMS destination MBean attributes*

Attribute	Description
DataThroughput	A read-only attribute showing the throughput of the destination in terms of JMS message data processes since the `InstrumentationStartTime`.
MessageInProgress	A read-only attribute showing the number of JMS messages being currently processed for the destination.
DataInProgress	A read-only attribute showing the amount of JMS message data being currently processed for the destination.
MaxMessageSize	A controllable attribute that specifies the maximum size, in bytes, allowed for the destination.
MaxConsumers	A controllable attribute that specifies the maximum number of consumers that can connect to the destination.
DurableConsumersAllowed	A controllable attribute that specifies if the destination allows durable consumers
MaxUnconsumedMessages	A controllable attribute that specifies the maximum number of unconsumed JMS messages the destination can store.
MaxUnconsumedData	A controllable attribute that specifies the maximum amount of unconsumed JMS message data, in bytes, the destination can store.

Selecting a Persistent Store Implementation

Overview

Currently, Orbix supports four persistent store configurations. It defaults to work with the Cloudscape database installed with Orbix. It can also work with Oracle using either the Merant SequeLink JDBC drivers which are shipped with Orbix or the Oracle JDBC drivers which are shipped with Oracle. JMS also supports an implementation that operates purely in memory and is transient for testing purposes.

Cloudscape

To configure JMS to use Cloudscape as its persistent store you must make the following configuration entries:

```
persistence:message_store="Cloudscape";
persistence:jdbc:driver="COM.cloudscape.core.JDBCDriver";
persistence:jdbc:url="jdbc:cloudscape:data;create=true";
```

Note: Cloudscape is the default configuration.

Oracle with Merant SequeLink JDBC drivers

To configure JMS to use Oracle, with the Merant SequeLink JDBC driver, as its persistent store you must make the following configuration entries:

```
persistence:message_store="Oracle";
persistence:jdbc:driver="com.iona.jms.store.jdbc.SequeLinkDriver
    Wrapper";
persistence:jdbc:url="jdbc:iona:sequelink://host:port;[Oracle];
    Database=sid";
persistence:jdbc:user="oracle_username";
persistence:jdbc:password="oracle_password";
```

You must also ensure that the Merant SequeLink JDBC driver's jar file is in the JMS CLASSPATH. For a default windows installation, the jar file would be located in *install_dir*\shared\lib\sljc_brabd-6.1\sljc_brand.jar.

Oracle with Oracle JDBC drivers

To configure JMS to use Oracle, with Oracle's JDBC drivers, as its persistent store you must make the following configuration entries:

```
persistence:message_store="Oracle";
persistence:jdbc:driver="oracle.jdbc.pool.OracleConnectionPoolDa
    taSource";
persistence:jdbc:url="jdbc:oracle:thin:@host:port:sid";
persistence:jdbc:user="oracle_username";
persistence:jdbc:passowrd="oracle_password";
```

You must also ensure that the Oracle JDBC driver's jar file is in the JMS CLASSPATH.

Pure memory store

To configure JMS to use a purely in-memory persistent store implementation you must make the following configuration entries:

```
persistence:message_store="Memory";
persistence:jdbc:driver="";
persistence:jdbc:url="";
```

WARNING: This implementation is transient and is only suitable for testing purposes. It is not suitable for deployment environments because it does not provide the reliability guarantees for re-delivery of unacknowledged messages as mandated by the JMS specification.

Running JMS Clients

JMS clients initialize the Orbix ORB silently and therefore do not pass along command line arguments to the ORB. This requires that you pass any ORB command line arguments as Java system properties.

Specifically, you must be sure to pass your domain name and configuration directories to the ORB. For example, to run the point-to-point demo's sender client you would type

```
java -DORBdomain_name=name -DORBconfig_domains_dir=dir
    demos.jms.pointToPoint.Sender
```

The following would not work:

```
java demos.jms.pointToPoint.Sender -ORBdomain_name name
    -ORBconfig_domains_dir dir
```

Part 6

The JMS-Notification Bridge Service

In this part

This part contains the following chapters:

JMS-Notification Message Translation

The JMS-Notification bridge translates messages between JMS and the notification service based on OMG specified standards.

In this chapter

This chapter discusses the following topics:

JMS Message to Notification Event

Overview

JMS messages are translated into structured events. The JMS header properties which define lifetime, priority, and persistence are mapped to the corresponding QoS properties in the variable header of the structured event. All other JMS header properties are stored in the filterable data portion of the structured event. The method used to translate the JMS message body into the structured event body depends on the type of JMS message being translated.

Figure 14 shows how a JMS message is mapped to a structured event that a notification service consumer can understand.
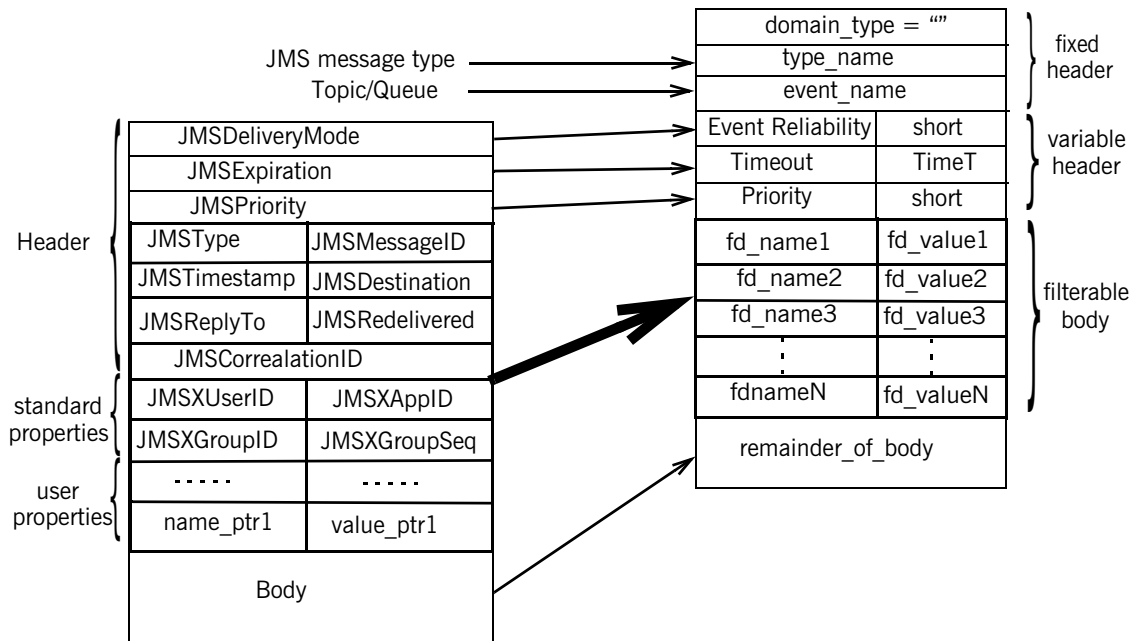
Figure 14: *JMS message to structured event mapping*

Structured event fixed header data   A structured event's fixed header fields are mapped as follows:

domain_name

The `domain_name` field is set to an empty string.

type_name

The `type_name` field is set to indicate the JMS message's type. The message's type specifier is prefixed with the percent (`"%"`) character. For example, a JMS text message would have a `type_name` of `"%TextMessage"`.

event_name

The `event_name` field is set to the name of the JMS destination from which the message was forwarded.

Persistence, lifetime, and priority property mapping   A JMS message's header uses three fields to specify a message's persistence, lifetime, and priority. These fields are mapped directly to notification service QoS properties in the structured event variable header. They are mapped as follows:

JMSDeliveryMode

The `JMSDeliveryMode` field of the JMS message header is mapped to the `EventReliability` QoS property and is set in the structured event's variable header field. If the `JMSDeliveryMode` is specified as `Persistent`, `EventReliability` is set to `Persistent`. All other `JMSDeliveryMode` settings are mapped to `BestEffort`.

JMSExpiration

The `JMSExperation` field of the JMS message header is mapped to the `Timeout` QoS property. The value is converted from milliseconds, JMS units for message timeout, to units of 100 nanoseconds, the notification service's units for message timeout.

JMSPriority

The `JMSPriority` field of the JMS message header is mapped to the `Priority` QoS property.

Other JMS header fields   The remaining fields of the JMS message header have no direct mapping into QoS property settings in a structured event's header field. Therefore, the remaining fields are mapped into the filterable data section of the structured

281

event. They are inserted into the filterable data section as name-value pairs using the `CosNotification::PropertySeq` data structure. JMS header fields with null values are omitted upon translation.

Optional JMS property fields

A JMS message can contain a number of optional property fields to further specify the message's origin and content. These properties, if specified, are placed into the structured event's filterable body along with the JMS header fields. They are also inserted using the `CosNotificatioin::PropertySeq` data structure.

JMS Message body

The body of a JMS message is inserted into the `remainder_of_body` portion of the structured event as a `CORBA::Any`. JMS message bodies are mapped differently depending on the type of JMS message body passing through the bridge. The different message types are mapped as follows:

TextMessage

A JMS `TextMessage` consists of a Java `String`. The data is inserted into the `remainder_of_body` by insertiting the `String` in a `CORBA::Any` as a wide string.

StreamMessage

A JMS `StreamMessage` consists of a stream of Java primitive types. The data in the stream is inserted into the `remainder_of_body` by translating each piece of data using the standard IDL to Java mapping and encapsulating the stream into a `CORBA::AnySeq`.

MapMessage

A JMS `MapMessage` consists of a sequence of name-value pairs where the name is a Java `String` and the value is a Java primitive type. The data is inserted into the `remainder_of_body` by translating the sequence of name-value pairs into a `CosNotification::PropertySeq` data structure. The Java primitive types are translated using the standard Java to IDL mapping.

BytesMessage

A JMS `BytesMessage` consists of uninterpreted stream data from either a Java `DataInputStream` or a Java `DataOutputStream`. The data is inserted into the `remainder_of_body` as an untranslated IDL `OctetSeq`.

ObjectMessage

A JMS `ObjectMessage` consists of a Java object that supports the `Serializable` interface. The data is inserted into the `remiander_of_body` as an `OctetSeq`. The data in the `OctetSeq` can be reconstructed by a notification consumer, but it is up to the consumer's developer to implement the reconstruction process.

Notification Event to JMS Message

Overview

All styles of notification events, Any, Sequence, and Structured, are treated as structured events when being mapped to a JMS message. The mapping of Any and Sequence events follows the standard mapping specified by the OMG. The QoS properties `EventRelaibility`, `Timeout`, and `Priority` are mapped to the corresponding fields in the JMS message header. THe remainder of the variable header fields and the filterable body are mapped into the user defined properties section of the JMS message as name-value pairs. The `reamiander_of_body` portion of the structured event is mapped to the JMS message body. The JMS message is assigned a `JMSType` of `StructuredEvent`. The remaining header and property fields of the JMS message are filled in by the bridge using default values.

Figure 15 shows how a structured event is mapped to a JMS message.
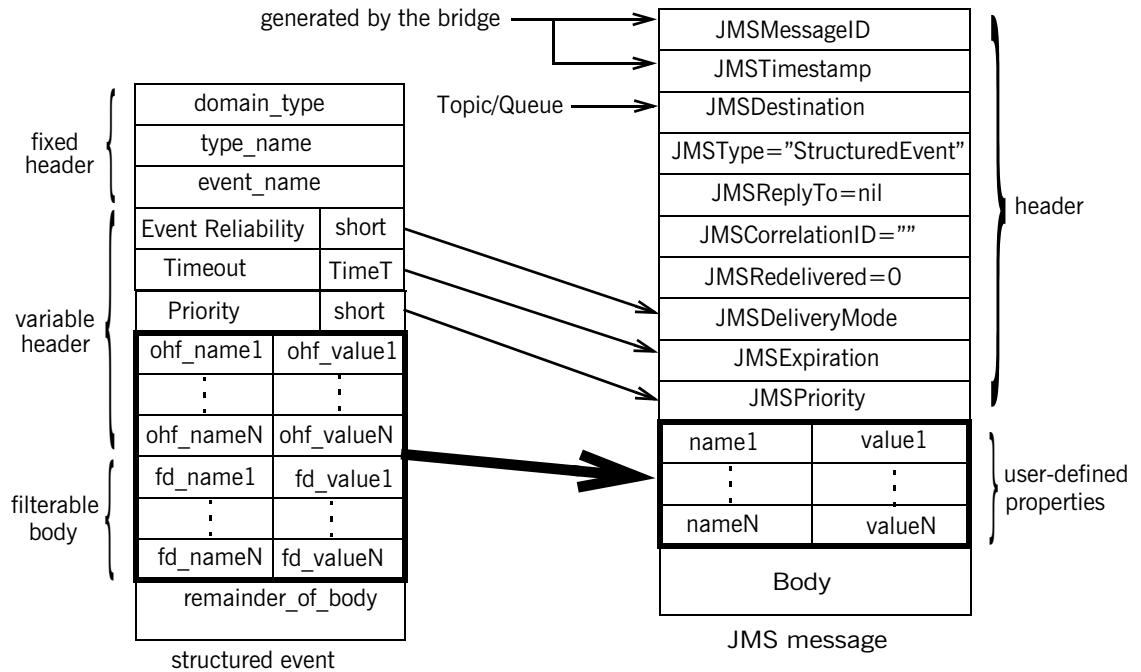
Figure 15: *Structured event to JMS message mapping*

QoS properties

The notification service properties EventReliability, Timeout, and Priority are mapped to the JMSDeliveryMode, the JMSExpiration, and the JMSPriority fields in the JMS message header.

EventReliability

If the EventReliability property is set in the event's variable header, the value is mapped to the JMSDeliveryMode field. If the EventReliability property is not set, JMSDeliveryMode is set to Persistent.

Timeout

If the Timeout property is set in the event's variable header, the value is mapped to the JMSExpiration field. The value is converted from units of 100 nanoseconds into milliseconds. If the Timeout property is not set, JMSExpiration is set to unlimited.

Priority

If the Priority property is set in the event's variable header, the value is mapped to the JMSPriority field. If Priority is not set, JMSPriority is set to 4.

Remainder of the JMS header

The remainder of the JMS message header fields are filled in by the bridge. Table 18 shows how the remaining header fields are filled in.

Table 18: *JMS Message Header Completion*

JMS header field	Value
JMSMessageID	A unique key prefixed by 'ID'.
JMSTimestamp	The time that the message was passed to JMS for delivery.
JMSDestination	The name of the Topic or Queue to which the message is being sent.
JMSType	'StructuredEvent'
JMSReplyTo	nil
JMSCorrelationID	''
JMSRedelivered	0

Structured event fixed header

The three fields of the structured events fixed header are converted into name value pairs and inserted in the JMS message as user defined properties. The names of the properties are prefixed by a '$' and the values are mapped to Java String. For example the domain_name field would be mapped to the JMS property $domain_name.

286

Remaining variable header fields   Any properties, other than the QoS properties, set in the structured event's variable header are converted into name value pairs and mapped to user defined properties in the JMS message.

The name of the event property is prefixed with a `'$'` when mapped to its corresponding JMS property. The value of the event property is mapped to the corresponding Java primitive type. For example, an event property `score` would be mapped to the JMS property `$score`.

Filterable data   The structured event's filterable data is mapped to the user defined properties section of the JMS message. Each name value pair in the filterable data is mapped to a JMS user defined property using the same mapping used for the variable header fields.

Event body   How the structured event's `remainder_of_body` is mapped to the JMS message body depends upon the complexity of the data packaged into the `CORBA::Any`.

Table 19 shows how the data in the `remiander_of_body` is mapped into the JMS message body.

Table 19: *CORBA::Any to JMS message mapping*

CORBA::Any	JMS message body
IDL basic types	Each element maps to a java primitive using the standard IDL to Java mapping. The data is inserted into the JMS message as a `StreamMessage` body.
Single string	Inserted directly into the JMS message body as a `String`.
`PropertySeq`	Each property is mapped to a name-value pair where the values are mapped to the corresponding Java primitive using the standard IDL to Java mapping. The data is inserted into the JMS message as a `MapMessage` body.

Table 19: *CORBA::Any to JMS message mapping*

CORBA::Any	JMS message body
`OctetSeq`	The data is mapped directly into an unfiltered byte stream and inserted as a `ByteMessage` body.
User constructed types	The data is mapped directly into an unfiltered byte stream and inserted as a `ByteMessage` body. The burden of reconstructing the data type is left to the JMS consumer.

Managing the JMS-Notification Bridge Service

The JMS-Notification bridge service is a light weight, easy to use service that can be managed using command line tools or through programatic interfaces.

In this chapter

This chapter discusses the following topics:

Configuring the Bridge Service

Configuring the environment

Once the notification service and JMS are configured and deployed into your configuration, the bridge is automatically configured and deployed.

Administrative properties

The bridge has only one configuration property:

`jms_notify_bridge:endpoint_admin_name`

This property sets the name of the notification service endpoint admin object. This property does not need to be set unless you are running more than one notification service that is to be bridged.

Optimization

The bridge is optimized by optimizing JMS and the notification service. There are no configuration variables that directly effect the performance of the bridge itself.

Running the Bridge Service

Launching the service

To launch the JMS-Notification bridge you need to launch the notification service, the JMS broker, and have a valid license to use the bridge service. The administrative services for the bridge are handled by the JMS broker, so there are no separate services that need to be launched.

For information on starting the notification service, see "Running the Notification Service" on page 151.

For information on starting JMS, see "Running JMS" on page 266.

Creating bridges

You can create a bridge in one of three ways:

- using itadmin.
- programatically.

Managing the Bridge Service with itadmin

Overview

Iona's command line admin tool, `itadmin`, can be used to create, manage and monitor bridges and their endpoints.

Creating a bridge

A bridge can be created using the following itadmin command:

```
itadmin bridge create -source_admin <IOR | INIT_REF_KEY>
                      -source_type <topic | queue | channel>
                      -source_name <source name>
                      -sink_admin <IOR | INIT_REF_KEY>
                      -sink_type <topic | queue | channel>
                      -sink_name <sink name>
                      <bridge name>
```

The bridge create command has the following arguments:

`source_admin`	The IOR or initial reference of the administrative object used to connect to the message source. To use the default notification endpoint admin use `"IT_NotificationEndpointAdmin"`; to use the default JMS endpoint admin use `"IT_JMSEndpointAdmin"`.
`source_type`	The type of object that will be passing messages into the bridge. It can take one of three values:
	topic if the messages will originate from a JMS topic.
	queue if the messages will originate from a JMS queue.
	channel if the messages will originate from a notification channel.
`source_name`	The name of the object that will be passing messages into the bridge.

sink_admin   The IOR or initial reference of the administrative object
                    used to connect to where messages are being forwarded.
                    If the message source is a notification channel, the
                    message sink should be a JMS Destination. To use the
                    default notification admin use
                    "IT_NotificationEndpointAdmin"; to use the default
                    JMS admin use "IT_JMSEndpointAdmin".

sink_type   The type of object that will be receiving messages from
                    the bridge. It can take one of three values:

                    topic if the messages are being forwarded to a JMS topic.

                    queue if the messages are being forwarded to a JMS
                    queue.

                    channel if the messages are being forward to a
                    notification channel.

sink_name   The name of the object that will receive messages from
                    the bridge.

bridge name   The name of the bridge. This must be a unique string
                    value that will be used to identify this bridge.

For example, to create a bridge, sports_bridge, from a notification channel,
sports_channel, to a JMS topic, sports_topic, you could use the following
command:

```
itadmin bridge create -source_admin
    "IT_NotificationEndpointAdmin"
                        -source_type channel
                        -source_name sports_channel
                        -sink_admin "IT_JMSEndpointAdmin"
                        -sink_type topic
                        -sink_name sports_topic
                        sports_bridge
```

When a bridge is created it is in stopped state and cannot begin passing
messages until it is explicitly started.

Controlling the flow of messages through a bridge

A bridge can either be started, suspended, or stopped. If a bridge is started, messages are forwarded through the bridge. If the bridge is suspended, messages are collected at the source of the bridge, but the messages are not forwarded until the bridge is restarted. If the bridge is stopped, messages are not forwarded by the bridge.

To start the flow of messages through a bridge use the following command:

```
itadmin bridge start <bridge name>
```

To suspend the flow of messages through a bridge use the following command:

```
itadmin bridge suspend <bridge name>
```

To stop the flow of messages through a bridge use the following command:

```
itadmin bridge stop <bridge name>
```

Monitoring bridges

itadmin provides commands for discovering and displaying the status of bridges and their endpoints.

To list all of the instantiated bridges in a deployment use the following command:

```
itadmin bridge list
```

To display the status of a bridge use the following command:

```
itadmin bridge show <bridge name>
```

Monitoring bridge endpoints

To display an endpoint admin's name and the type of endpoints it supports use the following command:

```
itadmin endpoint_admin show <IOR | INIT_REF_KEY>
```

To list the endpoints associated with an endpoint admin use the following command:

```
itadmin endpoint list <-source | -sink> -admin <IOR |
    INIT_REF_KEY>
```

You need to select whether you wish to list the source endpoints or the sink endpoints associated with the specified admin.

To display the status and attributes of a particular endpoint use the following command:

```
itadmin show <-source | -sink> -admin <IOR | INIT_REF_KEY>
    <bridge name>
```

Destroying bridges

itadmin provides commands for destroying endpoints and bridges.

To destroy an endpoint use the following command:

```
itadmin endpoint destroy <-source | -sink> -admin <IOR |
    INIT_REF_KEY> <bridge name>
```

You need to specify whether the endpoint is a message source or a message sink and what type of admin object with which it is associated.

To destroy a bridge use the following command:

```
itadmin bridge destroy <bridge name>
```

Managing the Bridge Service Programatically

Overview

The JMS-notification bridge provides a APIs for both JMS and notification clients to use in creating and managing bridges. These APIs are specified in the following IDL modules:

- `IT_MessagingBridgeAdmin`
- `IT_MessagingBridge`
- `IT_NotifyBridge`

Actions

Applications that programatically manage bridges perform the following actions:

- Get a `BridgeAdmin` object which serves as a factory for bridges.
- Locate existing bridges.
- Get `MessagingEndpoint` objects.
- Create new bridges.
- Start message flow through a bridge.
- Stop the flow of messages through a bridge.
- Destroy bridges.

In this section

This section discusses the following topics:

Getting a BridgeAdmin

The `BridgeAdmin` interface, defined in `IT_MessagingBridgeAdmin`, is a bridge factory. You get an instance of the `BridgeAdmin` by using the standard CORBA call `resolve_initial_references()` with the key `"IT_MessagingBridge"` and narrowing the returned object.

Example 84 shows the code used to get a `BridgeAdmin`.

Example 84: *Getting a BridgeAdmin instance*

```
org.omg.CORBA.Object obj =
   orb.resolve_initial_references("IT_MessagingBridgeAdmin");
com.iona.messaging.IT_MessagingBridgeAdmin.BridgeAdmin
   bridge_admin =
   com.iona.messaging.IT_MessagingBridgeAdmin.BridgeAdminHelper.
   narrow(obj);
```

Getting a Bridge

Operations

The `BridgeAdmin` interface provides three operations for getting a bridge:

```
// IDL in IT_MessagingBridgeAdmin::BridgeAdmin

Bridge create_bridge(in BridgeName    bridge_name,
                     in EndpointInfo source,
                     in EndpointInfo sink);
raises (InvalidEndpoint, BridgeAlreadyExists,
        BridgeNameAlreadyExists, CannotCreateBridge);

Bridge get_bridge(in BridgeName bridge_name)
raises (BridgeNotFound);

Bridge find_bridge(in EndpointInfo source,
                   in EndpointInfo sink,
                   out Bridgename  bridge_name)
raises (BridgeNotFound);
```

Creating endpoints

The `create_bridge()` and `find_bridge()` operations require that you first specify both the source and sink endpoints of the bridge using an `IT_MessagingBridgeAdmin::EndpointInfo` element. `EndpointInfo` contains three fields:

admin specifies the `EndpointAdmin` to which the endpoint will be associated. The `EndpointAdmin` is obtained by calling `resolve_initial_references()` using `"IT_NotificationEndpointAdmin"` to obtain an endpoint in the notification service or `"IT_JMSEndpointAdmin"` to obtain an endpoint in JMS.

type specifies the type of the endpoint. It can take one of three values:

- ♦ `IT_MessagingBridge::JMS_TOPIC`
- ♦ `IT_MessagingBridge::JMS_QUEUE`
- ♦ `IT_MessagingBridge::NOTIFY_CHANNEL`

name specifies the name of the messaging object to which the endpoint is associated.

The code in Example 85 creates a source endpoint for connecting to a JMS topic.

Example 85: *Creating an endpoint*

```java
// Java
import com.iona.messaging.IT_MessagingBridge.*;
import com.iona.messaging.IT_MessagingAdmin.*;

EndpointInfo endpt = new EndpointInfo();

org.omg.CORBA.Object obj =
    orb.resolve_initial_references("IT_JMSEndpointInfo");

endpt.admin = EndpointAdminHelper.narrow(obj);
endpt.name = "sports_topic";
endpt.type = EndpointType.JMS_TOPIC.value;
```

1 `org.omg.CORBA.Object obj =`

2 `endpt.name = "sports_topic";`

3 `endpt.type = EndpointType.JMS_TOPIC.value;`

The previous example does the following:

1. Get a reference to the JMS `EndpointAdmin` by calling `resolve_initial_references()` and narrowing the returned object reference.

2. Specify the name of the messaging object to which the endpoint is going to be associated.

3. Specifies that the endpoint will connect to a JMS topic.

Creating a bridge

You create new unidirectional bridges by calling the `BridgeAdmin`'s `create_bridge()` operation. The operation takes three parameters:

bridge_name is a unique string identifier for the new bridge. If the specified name is already used by another bridge the operation will raise the `IT_MessagingBridge::BridgeNameAlreadyExists` exception.

source specifies the endpoint that connects to the source of the messages being forwarded through the bridge. It is specified as an `EndpointInfo`.

sink specifies the endpoint that connects to the destination of the messages being forwarded through the bridge. It is specified as an `EndpointInfo`.

For information on specifying endpoints see "Creating endpoints" on page 298.

Before the newly created bridge will begin forwarding messages, it must be explicitly started by calling the `start()` operation on it.

Finding existing bridges

You can get a reference to an already existing bridge by either specifying the bridge's name or by specifying the bridge's endpoints.

The `BridgeAdmin`'s `get_bridge()` operation allows you to discover an existing bridge using its unique name. The operation will return a reference to the specified bridge if it exists; otherwise it will raise `IT_MessagingBridgeAdmin::BridgeNotFound`.

If you do not know the bridge's unique name, but do know the bridge's endpoints you can use the `BridgeAdmin`'s `find_bridge()` operation to get a reference to the bridge. `find_bridge()` takes the source and sink `EndpointInfo` for the bridge and will return a reference to the bridge if it exists. If the bridge does not exist it will raise `IT_MessagingBridgeAdmin::BridgeNotFound`.

Managing Message Flow Through a Bridge

Operations

The IT_MessagingBridgeAdmin::Bridge interface defines three operations to control the flow of messages through a bridge:

start() begins the flow of messages through the bridge. The bridge will forward messages until another call stops or suspends the flow of messages.

Note: When a bridge is first created it must be explicitly started before messages can flow through it.

suspend() stops the flow of messages through the bridge. The bridge will continue to queue messages for forwarding when the bridge is restarted.

stop() stops the flow of messages through the bridge. The bridge will not continue to accept any messages for forwarding until it is restarted.

These operations take no parameters.

Example

The code in Example 86 gets a bridge named "sports_bridge" and starts the flow of messages through it.

Example 86: *Starting a bridge*

```
// Java
try
{
  com.iona.messaging.IT_MessagingBridgeAdmin.Bridge bridge=
    bridge_admin.get_bridge("sports_bridge");
}
catch(com.iona.messaging.IT_MessagingBridgeAdmin.BridgeNotFound)
{
  // handle the exception
}

bridge.start();
```

Destroying a Bridge

Operation

You destroy a bridge by calling its `destroy()` operation. The bridge's `destroy()` operation frees all resources used to maintain the bridge.

Example

The code in Example 87 gets the bridge named "sports_bridge" and destroys it.

Example 87: *Destroying a bridge*

```
// Java
try
{
  com.iona.messaging.IT_MessagingBridgeAdmin.Bridge bridge=
    bridge_admin.get_bridge("sports_bridge");
}
catch(com.iona.messaging.IT_MessagingBridgeAdmin.BridgeNotFound)
{
  // handle the exception
}

bridge.destroy();
```

Index