MICRO FOCUS

Orbix 6.3.11

Enterprise Messaging Guide: Java

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

1/31/19

Contents

Preface

Specification compliance

The Orbix Notification Service is a full implementation of the notification service as specified by the Object Management Group.

The Orbix Telecom Log Service is a full implementation of the telecom log service a specified by the Object Management Group.

All CORBA messaging services comply with the following specifications:

- CORBA 2.6
- GIOP 1.2 (default), 1.1, and 1.0

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 C++.

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*.

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.

Contacting Micro Focus

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/orbix/orbix-6.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx. (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Part I

Messaging Service Technologies

Overview

Orbix provides enterprise messaging technology through the CORBA notification service and the CORBA telecom log service.

In this part

This part contains the following chapters:

CORBA Messaging Technologies

The architecture 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.

Event Service

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.

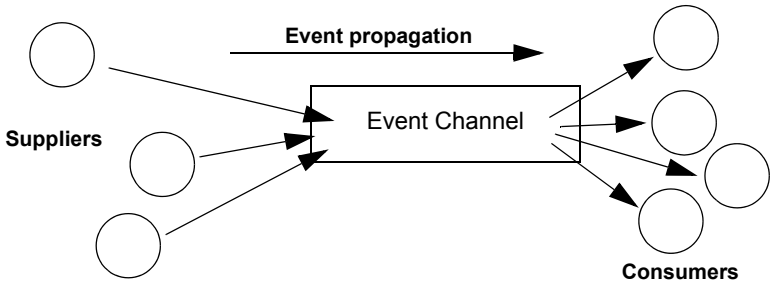

Figure 1: *Suppliers and Consumers Communicating through an Event Channel*

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.

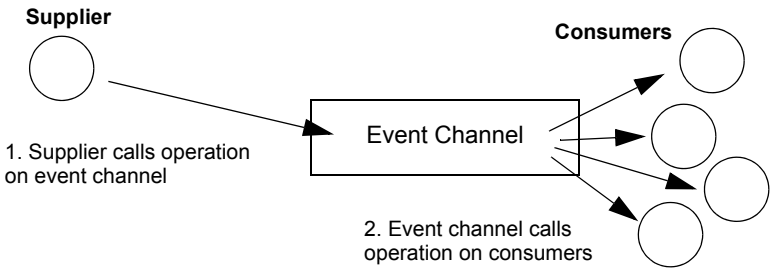

Figure 2: *Event Propagation in a CORBA System*

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.

Feature	Description
Multicast event delivery	Groups of consumers can subscribe to events and receive them using UDP multicast protocol, which keeps network traffic to a minimum.

Note: The CORBA notification service is integrated with the other Orbix services. However, it is not designed for use with the Object Transaction Service (OTS).

For more information on the CORBA notification service, see "The Notification Service" on page 15

Telecom Log Service

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.

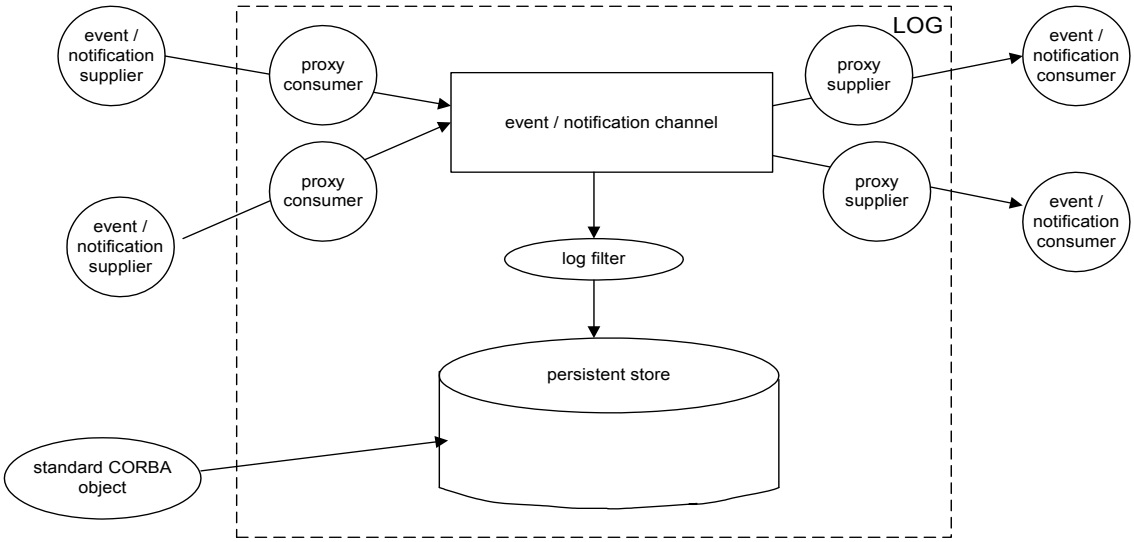

Figure 3: *Log service configuration*

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 "The Telecom Log Service" on page 109.

Event Communication

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.

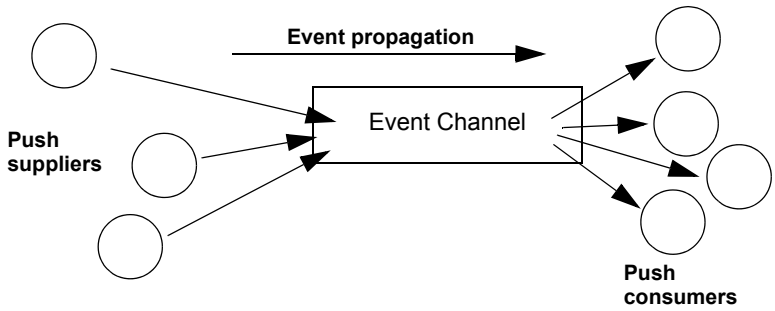

Figure 4: *The Push Model of Event Transfer*

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.

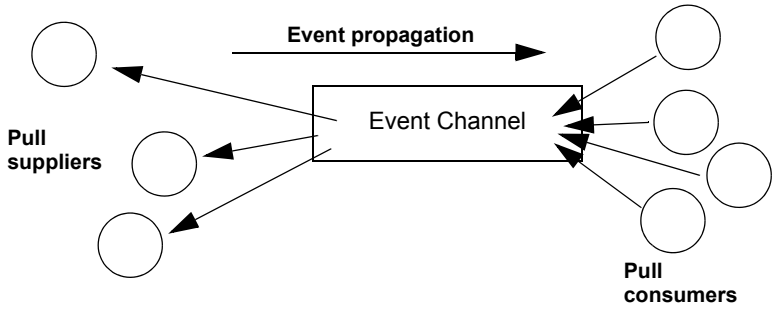

Figure 5: *Pull Model Suppliers and Consumers Communicating through an Event Channel*

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.

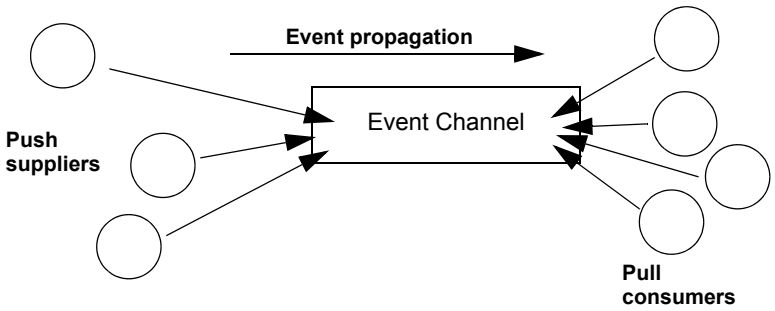

Figure 6: *Push Suppliers and Pull Consumers Communicating through an Event Channel*

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.

Part II

The Notification Service

In this part

This part contains the following chapters:

Developing Suppliers and Consumers

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

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.

Obtaining an Event Channel

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.

OMG Operations

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

```
// 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*

```
// C++
CosNotifyChannelAdmin::EventChannel_var ec;
CosNotifyChannelAdmin::ChannelID id;
CosNotification::QoSProperties init_qos(0);
CosNotification::AdminProperties init_admin(0);

1 CORBA::Object_var obj =
    orb->resolve_initial_references("NotificationService");
  IT_NotifyChannelAdmin::EventChannelFactory_var factory =
    IT_NotifyChannelAdmin::EventChannelFactory::_narrow(obj);
2 try
    {
      ec = factory->create_named_channel("EventChannel", init_qos,
                                         init_admin, id);
    }
```

Example 1: *Obtaining an Event Channel*

```
3  catch (IT_NotifyChannelAdmin::ChannelAlreadyExists&)
     {
4      // Channel has been previously created, so find it
       try
         {
           ec = factory->find_channel("EventChannel", id);
         }
       catch (CosNotifyChannelAdmin::ChannelNotFound&)
         {
           cerr << "Couldn't create or find the event channel" <<
                endl;
           exit(1);
         }
       catch (CORBA::SystemException& se)
         {
          cerr << "System exception occurred during find_channel:
    "
                << se << endl;
          exit(1);
         }
     } // catch ChannelAlreadyExists
```

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.

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)*

```
// C++
#include <omg/CosNotification.hh>
#include <omg/CosNotifyChannelAdmin.hh>
#include <omg/CosNotifyCommS.hh>

class NotifyPushSupplier_i :
  public virtual POA_CosNotifyComm::StructuredPushSupplier
{
  public:
    NotifyPushSupplier_i()
    { }
    ~NotifyPushSupplier_i()
    { }

  // ...
}
```

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

```
// ...

// client supplier program
int main(int argc, char *argv[])
{
  // ...
  // ORB and POA activation not shown
  // ...

  supplier = new NotifyPushSupplier_i;

  // ...
}
```

Connecting to a Channel

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:

```
CosNotifyChannelAdmin::SupplierAdmin_var sa =
  channel->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 "Traversing Multiple Filters in a Channel" on page 64).

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 "Notification Service Properties".
- 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 59), 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 56) 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*

```
// C++
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ClientType ctype =
  CosNotifyChannelAdmin::STRUCTURED_EVENT;

try
{
  CosNotifyChannelAdmin::ProxyConsumer_var obj =
    sa->obtain_notification_push_consumer(ctype,
   proxy_id);
}
catch(CosNotifyChannelAdmin::AdminLimitExceeded err)
{
// handle the exception
}

CosNotifyChannelAdmin::StructuredProxyPushConsumer_ptr
   ppc =
CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narr
   ow(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*

```
// C++
// proxy ppc and PushSuplier supplier obtained previously
try
 {
   ppc->connect_structured_push_supplier(supplier)
 }
catch (CosEventChannelAdmin::AlreadyConnected.value ac)
  {
   // Handle the exception
  }
catch (CORBA::SystemException& se)
  {
    cerr << "System exception occurred during connect: "
   <<
         se << endl;
    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 7.

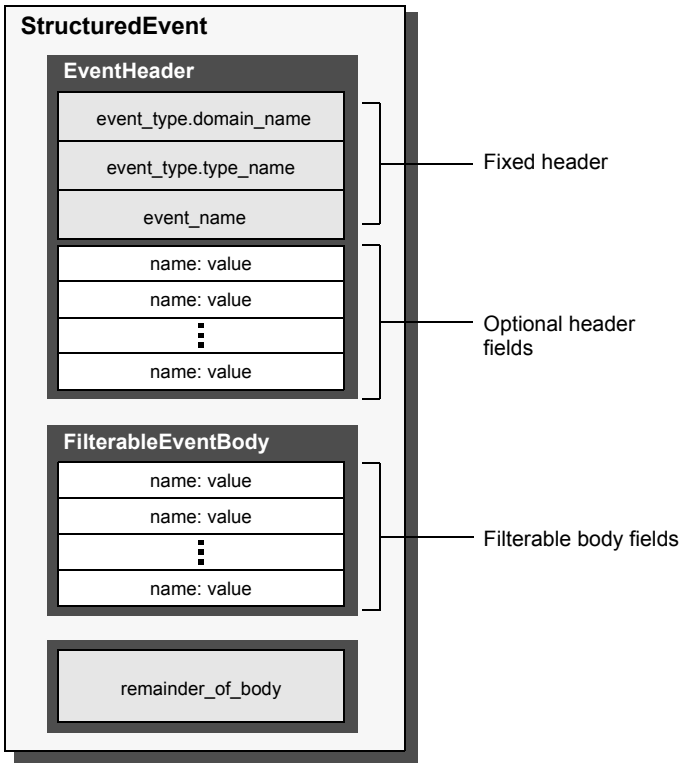

Figure 7: *Structured Event Components*

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 ("Event Filtering") and quality-of-service properties ("Notification Service Properties").

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*

```
//C++
CosNotification::StructuredEvent se;
se.header.fixed_header.event_type.domain_name =
  string_dup("SportsNews");
se.header.fixed_header.event_type.type_name =
  string_dup("BaseballResults");
se.header.fixed_header.event_name = string_dup("");
se.header.variable_header.length(1);
se.header.variable_header[0].name =
  string_dup(CosNotification::Priority);
se.header.variable_header[0].value <<= 0;
se.filterable_data.length(0);
```

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*

```
// C++
// proxy consumer and event message already obtained
try
  {
    proxy->push_structured_event(se);
  }
catch (CORBA::SystemException& sysex)
  {
    cerr << "System exception occurred during push: " <<
         sysex << endl;
    exit(1);
  }
catch (CORBA::Exception&)
  {
    cerr << "Unknown exception occurred during push" <<
   endl;
    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 55). The interval between retries is specified by the `PullInterval` property (see "PullInterval" on page 56).

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*

```
// C++
CosNotification::StructuredEvent*
NotifyPullSupplier_i::try_pull_structured_event(boolean
                                                has_event)
throw(CORBA::SystemException)
{
  boolean has_scores = false;

  has_event = false;
  // check if any baseball scores are available
  string scores = get_latest_scores(has_scores));

    CosNotification::StructuredEvent *event =
      new CosNotification::StructuredEvent();

    if (has_scores)
    {
      event->header.fixed_header.event_type.domain_name =
        string_dup("SportsNews");
      event->header.fixed_header.event_type.type_name =
        string_dup("BaseballResults");

      // no filterable data
     event->header.variable_header.length(0);
     event->filterable_data.length(0);

      event->remainder_of_body <<= scores;
      has_event = true;
    }
    return(event);
}
```

Disconnecting From the Event Channel

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.

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)*

```
// C++
// client consumer definitions
#include <omg/CosNotification.hh>
#include <omg/CosNotifyChannelAdmin.hh>
#include <omg/CosNotifyCommS.hh>
```

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

```
class NotifyPushConsumer_i :
 public virtual POA_CosNotifyComm::StructuredPushConsumer
{
  public:
// ...

  void push_structured_event(const

   CosNotification::StructuredEvent);
    // ...
}
// client consumer program
int main(int argc, char *argv[])
{
  // ORB and POA activation not shown
  // ...
  consumer = new NotifyPushConsumer_i;

  // ...
}
```

Connecting to the Channel

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:

```
CosNotifyChannelAdmin::ConsumerAdmin_var 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 64).

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 "Notification Service Properties".
- 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 "Event Filtering".

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 56) 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*

```
// C++
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ClientType ctype =
  CosNotifyChannelAdmin::STRUCTURED_EVENT;

try
{
  CosNotifyChannelAdmin::ProxySupplier_var obj =
    ca->obtain_notification_push_supplier(ctype,
   proxy_id);
}
catch(CosNotifyChannelAdmin::AdminLimitExceeded err)
{
 // handle the exception
}

CosNotifyChannelAdmin::StructuredProxyPushSupplier_ptr
   pps =

   CosNotifyChannelAdmin::StructuredProxyPushSupplier::_nar
   row(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*

```
// C++
// Proxy pps and PushConsumer consumer obtained
   previously
try
  {
    pps->connect_structured_push_consumer(consumer)
  }
catch (CosEventChannelAdmin::AlreadyConnected ac)
  {
     cerr << "Already connected to channel." << endl;
     exit (1);
  }
catch (CORBA::SystemException& se)
  {
    cerr << "System exception occurred during connect: "
         << se << endl;
    exit(1);
  }
```

Obtaining Event Messages

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 Operation*

```
// C++
void NotifyPushConsumer_i::push_structured_event
     (const CosNotification::StructuredEvent& event)
     throw(CORBA::SystemException)
{
  CORBA::String news_type;
  CORBA::String sports_type;
  CORBA::String scores;

  event.domain_name >> = news_type;
  event.type_name >>= sports_type;

  if (strcmp(news_type, "SportsNews") == 0)
    {
      event.remainder_of_body >>= scores;

      cout << "Current " << sports_type << "scores: " <<
   scores
           << endl;
    }
}
```

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 55). The interval between retries is specified by the `PullInterval` property (see "PullInterval" on page 56).

Example

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

Example 12: *Pulling Events*

```
// C++
CosNotification::StructuredEvent *event;
CORBA::ULong n;
boolean has_data = false;

try
  {
    event = proxy->try_pull_structured_event(has_data);
  }
catch (CosEventComm::Disconnected&)
  {
   cerr << "Disconnected exception occurred during pull"
  <<
        endl;
    exit(1);
  }
catch (CORBA::SystemException& se)
  {
    cerr << "System exception occurred during pull" <<
  endl;
    exit(1);
  }

if (has_data)
  {
    event->remainder_of_body >>= n;
    cout << "Received event number " << n << "using
  try_pull"
        << endl;
}
```

Disconnecting From the Event Channel

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.

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.

Property Types

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 48).

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
`OrderPolicy`		Y	Y	Y
`StopTime`	Y			

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

Property	Message	Proxy	Admin	Channel
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

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.

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` specifing 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*

```
//C++
// event channel chan obtained earlier

try
 {
    CosNotification::QoSProperties_var newQos =
      new CosNotification::QoSProperties();
    newQos->length(1);
    newQos[0].name =
      CORBA::string_dup(CosNotification::OrderPolicy);
     newQos[0].value <<= CosNotification::FifoOrder;
     chan->set_qos(newQos);
}
catch (CosNotification::UnsupportedQoS& unsupported)
  {
    // deal with exception
  }
catch (CORBA::SystemException& se )
  {
    cerr << "System exception occurred during << set_qos call.
             Error: " << se << endl;
  }
```

Setting a Structured Event's QoS Properties

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 see page 47.

Example

Example 14 sets a structured event's `Priority` property to 0.

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

```
//C++
CosNotification::StructuredEvent se;
se.header.fixed_header.event_type.domain_name =
  CORBA::string_dup("SportsNews");
se.header.fixed_header.event_type.type_name =
  CORBA::string_dup("BaseballResults");
se.header.fixed_header.event_name =
   CORBA::string_dup("");
se.header.variable_header.length(1);
se.header.variable_header[0].name =
  CORBA::string_dup(CosNotification::Priority);
se.header.variable_header[0].value <<= 0;
se.filterable_data.length(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 41), 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*

```
//C++
CosNotification::QoSProperties_var 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 44.

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 Message Properties*

```
// C++
// consumer proxy ppc previously obtained
try
  {
    CosNotification::QoSProperties_var reliableQos =
      new CosNotification::QoSProperties();
    reliableQos->length(1);
    reliableQos[0].name =
      CORBA::string_dup(CosNotification::EventReliability);
    reliableQos[0].value <<= CosNotification::Persistent;
    ppc->validate_event_qos(reliableQos);
  }
catch(CosNotification::UnsupportedQoS& unsupported)
  {
    cout << "Event persistence not allowed; event channel <<
             must be set to BestEffort. Error: unsupported \n";
  }
 catch(CORBA::SystemException& se)
  {
    cerr << "System exception occurred during <<
             validate_event_qos call. Error: " << se << endl;
 }
```

Property Descriptions

The following topics are discussed in this section:

- Reliability Properties
- Event Priority
- Event Queue Order
- Lifetime Properties
- Start Time Properties
- Undelivered Event Properties
- Discard Policy
- Sequenced Events Properties
- Proxy Push Supplier Properties
- Proxy Pull Consumer Properties
- RequestTimeout
- Channel Administration Properties

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:

	ConnectionReliability	
EventReliability	`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 50); 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 65).

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 65).

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 effectively 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

`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`. Micro Focus'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.

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`.

Forwarding Filters

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.

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*

```
// C++
// event channel obtained earlier
CosNotifyFilter::FilterFactory_var dff =
  channel->default_filter_factory();

CosNotifyFilter::Filter_var 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 70).

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*

```
// C++
CosNotification::EventTypeSeq event_types(1);
event_types.length(1);

1 event_types[0].domain_name =
    CORBA::string_dup("Orbix E2A Demos");
  event_types[0].type_name =
    CORBA::string_dup("Structured Notification Push Demo Event");

2 CosNotifyFilter::ConstraintExpSeq constraints(1);
  constraints.length(1);

  constraints[0].event_types = event_types;
3 constraints[0].constraint_expr =
    CORBA::string_dup("($EventNumber/2) ==
     (($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:

```
CosNotifyFilter::ConstraintInfoSeq* info =
  filter->add_constraints(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*

```
// C++
// event channel ca obtained earlier
CosNotifyFilter::FilterID fid;

// create structured push supplier proxy
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ClientType ctype =
  CosNotifyChannelAdmin::STRUCTURED_EVENT;

CosNotifyChannelAdmin::ProxySupplier_var obj =
  ca->obtain_notification_push_supplier(ctype, proxy_id);

CosNotifyChannelAdmin::StructuredProxyPushSupplier_ptr pps =
 CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(obj);

// add filter to proxy
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 64).

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 63.

Processing Events with Forwarding Filters

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 Figure 8 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

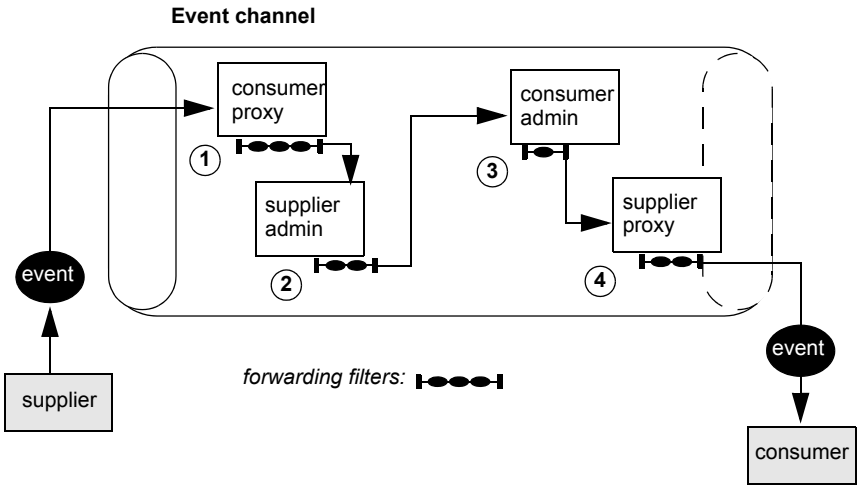

Figure 8: *Forwarding Filters Can Intercept an Event Message at Multiple Delivery Points*

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

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.

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.

Step	Action
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 69). 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*

```
// C++
// channel obtained earlier
CosNotifyFilter::FilterFactory_var dff =
  chan->default_filter_factory();
CORBA::Any default_value;

// set filter's default priority to 2
CORBA::Short value = 2;
default_value <<= value;
CosNotifyFilter::MappingFilter_var 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 71).
- 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: *Mapping Filter Constraints (Sheet 1 of 2)*

```
// C++
CosNotifyFilter::MappingConstraintPairSeq mapexp(2);
mapexp.length(2);

CosNotification::EventTypeSeq event_types(1);
event_types.length(1);

CosNotifyFilter::ConstraintExpSeq constraints(1);
constraints.length(1);
```

Example 21: *Mapping Filter Constraints (Sheet 2 of 2)*

```
// set up first constraint
event_types[0].domain_name = CORBA::string_dup("SportsNews");
event_types[0].type_name =
   CORBA::string_dup("BaseballResults");
constraints[0].event_types = event_types;
constraints[0].constraint_expr =
  CORBA::string_dup("($Priority < 100)");
mapexp[0].constraint_expression = constraints;
mapexp[0].result_to_set <<= (CORBA::Short) 100;

// set up second constraint
// domain_name is still SportsNews
event_types[0].type_name =
   CORBA::string_dup("FootballResults");
constraints[0].event_types = event_types;
constraints[0].constraint_expr =
  CORBA::string_dup("($Priority > 0)");
mapexp[1].constraint_expression = constraints;
mapexp[1].result_to_set <<= (CORBA::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:

```
CosNotifyFilter::MappingConstraintInfoSeq_var 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
structured_pps->priority_filter(Mapfilter);
```

Processing Events with Mapping Filters

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

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.

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 74). 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

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.

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 Filtering Constraint Language*

```
// C++
CosNotification::EventTypeSeq event_types(1);
event_types.length(1);

CosNotifyFilter::ConstraintExpSeq constraints(1);
constraints.length(1);

event_types[0].domain_name = CORBA::string_dup("SportsNews");
event_types[0].type_name =
   CORBA::string_dup("BaseballResults");
constraints[0].event_types = event_types;
constraints[0].constraint_expr =
  CORBA::string_dup("($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`:

```
CosNotification::EventTypeSeq event_types(0);
event_types.length(0);
```

- Initialize a single-element `EventTypeSeq` to empty strings:

```
CosNotification::EventTypeSeq event_types(1);
event_types.length(1);
event_types[0].domain_name = CORBA::string_dup("");
event_types[0].type_name = CORBA::string_dup("");
```

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

```
CosNotification::EventTypeSeq event_types(1);
event_types.length(1);
event_types[0].domain_name = CORBA::string_dup("*");
event_types[0].type_name = CORBA::string_dup("*");
```

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`:

```
CosNotification::EventTypeSeq event_types(1);
event_types.length(1);

// set up first contstraint
event_types[0].domain_name = CORBA::string_dup("*News");
event_types[0].type_name = CORBA::string_dup("*");
// ...
```

Referencing Filtered Data

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
```

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)
```

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

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
```

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.

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.

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

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

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 77).

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 78).

Example

Example 23 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 23: *Instantiating a Consumer for Multicast*

```
// C++
class NotifyConsumer_i :
public virtual POA_IT_NotifyComm::GroupPushConsumer
{
  public:
    NotifyConsumer_i();
    ~NotifyConsumer_i();

   virtual void push(const CORBA::Any &event)
   throw(CORBA::SystemException);

   virtual void disconnect_push_consumer()
   throw(CORBA::SystemException);

   virtual void
   connect(IT_NotifyChannelAdmin::GroupProxyPushSupplier_ptr
   proxy);

};

CORBA::ORB_var orb;
NotifyConsumer_i miop_consumer;

int main(int argc, char *argv[])
{
// ...

orb = CORBA::ORB_init(argc, argv, "egmiop_test");

// ...
}
```

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 78).

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 24 creates a POA with the correct policies. It must be run by every consumer wishing to join the endpoint group.

Example 24: *Creating a POA for an Endpoint Group (Sheet 1 of 2)*

```
// C++
// Create POAs for an endpoint group
#include <omg/CosNotifyChannelAdmin.hh>
#include <omg/CosNotifyCommS.hh>
#include <orbix/notify_channel_admin.hh>
#include <orbix/notify_commS.hh>

NotifyConsumer_i consumer = new NotifyConsumer_i;

CORBA::Object_var obj =
  orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa =
  PortableServer::POA::_narrow(obj);
PortableServer::POAManager_var poa_manager =
  root_poa->the_POAManager();
```

Example 24: *Creating a POA for an Endpoint Group (Sheet 2 of 2)*

```
// Create the policy set required for multicast references
CORBA::PolicyList policies(4);
policies.length(4);
CORBA::Any any;
any <<= "miop";
policies[0] =
  orb->create_policy(IT_CORBA::WELL_KNOWN_ADDRESSING_POLICY_ID,
                     any);
policies[1] =
  root_poa->create_lifespan_policy(PortableServer::PERSISTENT);
any <<= IT_PortableServer::DIRECT_PERSISTENCE;
policies[2] =

   orb->create_policy(IT_PortableServer::PERSISTENCE_MODE_POLIC
   Y_ID,
                     any);
policies[3] =

   root_poa->create_id_assignment_policy(PortableServer::USER_I
   D);

PortableServer::POA_var 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 25 shows how an endpoint group member might register with a group object reference.

Example 25: *Registering with a Group Object Reference (Sheet 1 of 2)*

```
1 // C++
PortableServer::ObjectId_var oid =
  PortableServer::string_to_ObjectId("miopConsumer");
```

Example 25: *Registering with a Group Object Reference (Sheet 2 of 2)*

```
2  multicast_poa->activate_object_with_id(oid, &consumer);
   consumer.myid(oid);


3  PortableServer::POAManager_var multicast_poa_manager =
     multicast_poa->the_POAManager();
   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

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 26 shows how to connect an endpoint group of `GroupPushConsumers` to a notification channel.

Example 26: *Connecting an Endpoint Group to an Event Channel (Sheet 1 of 2)*

```
// C++
CosNotifyChannelAdmin::EventChannel_var ec;
CosNotifyChannelAdmin::ChannelID id;
CosNotification::QoSProperties initqos(0);
CosNotification::AdminProperties initadmin(0);

main(int argv, char argc[])
{
// ...
1 CORBA::Object_var obj =
    orb->resolve_initial_references("NotificationService");
  IT_NotifyChannelAdmin::EventChannelFactory_var factory =
    IT_NotifyChannelAdmin::EventChannelFactory::_narrow(obj);


2 try
    {
      ec = factory->create_named_chanel("miop_channel",
                                        initqos, initadmin, id);
    }
  catch(CosNotifyChannelAdmin::ChannelAlreadyExists&)
    {
  // the channel already exists so try to find it
      try
        {
          ec = factory->find_channel("miop_channel", id);
        }
      catch(CosNotifyChannelAdmin::ChannelNotFound&)
        {
           cerr << "Cannot create or find notification channel"
           exit(1);
         }
    }


3 CosNotifyChannelAdmin::InterFilterGroupOperator op =
    CosNotifyChannelAdmin::AND_OP;
  CosNotifyChannelAdmin::AdminID id;
  CosNotifyChannelAdmin::ConsumerAdmin_var ca =
    channel->new_for_consumers(op, id);

4 CosNotifyChannelAdmin::ProxyID proxy_id;
  CosNotifyChannelAdmin::ClientType ctype =
    CosNotifyChannelAdmin::ANY_EVENT;
  CosNotifyChannelAdmin::ProxySupplier_var obj =
    ca->obtain_notification_push_supplier(ctype, proxy_id);
  IT_NotifyChannelAdmin::GroupProxyPushSupplier_var pps =
    IT_NotifyChannelAdmin::GroupProxyPushSupplier::_narrow(obj);
```

Example 26: *Connecting an Endpoint Group to an Event Channel (Sheet 2 of 2)*

```
5       try
          {
            pps->connect_group_any_push_consumer(consumer);
          }
        catch(CosEventChannelAdmin::AlreadyConnected)
          {
          // implementation left to developer
          }

    // ...
    }
```

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 36).

Filtering and Event Subscription

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 100).

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 59 and "Subscribing and Publishing" on page 89.

Disconnecting from an Event Channel

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.

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.

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.

Event Subscription

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.

Adding Forwarding Filters

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 93.) 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 59.

Example

Example 27 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 27: *Client Push Supplier*

```
\\ C++
class NotifyPushSupplier_i :
    public virtual POA_CosNotifyComm::StructuredPushSupplier
{
    struct SuppliedType
    {
        CORBA::String_var domain_name;
        CORBA::String_var type_name;
        CORBA::Boolean supply;
    };
    SuppliedType *m_supply_types;

    // Hard code the number of events suppplied
    CORBA::ULong NUM_TYPES_I_SUPPLY = 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:

```
CosNotification::EventTypeSeq
  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 93).

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 93).

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 `EventTypeSeq`, which contains all event types currently requested by consumers.

Example

In Example 28, 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 28: *Implementing* `obtain_subscription_types()`

```
//C++
// Initialize supply list with types of supplied sporting events
void
NotifyPushSupplier_i::init_supply_types()
{
    CORBA::ULong i;

1   m_supply_types = new SuppliedType[NUM_TYPES_I_SUPPLY];

    for (i = 0; i < NUM_TYPES_I_SUPPLY; i++)
    {
        m_supply_types[i].domain_name =
          CORBA::string_dup("Sports_News");
        m_supply_types[i].supply = IT_FALSE;
    }
    m_supply_types[0].type_name =
      CORBA::string_dup("Baseball_Results");
    m_supply_types[1].type_name =
      CORBA::string_dup("Football_Results");
    // ... other sporting event types
    m_supply_types[4].type_name =
      CORBA::string_dup("Tennis_Results");
```

Example 28: *Implementing* `obtain_subscription_types()`

```
2     // get list of types consumers are currently interested in
      CosNotification::EventTypeSeq_var types_to_supply;
      types_to_supply =
        m_proxy->obtain_subscription_types(ALL_NOW_UPDATES_ON);

3     // For each supplied event type which consumers want,
      // set its boolean flag to true
      CORBA::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 = IT_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 re-evaluate 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 29 the supplier updates the list of events that it can supply.

Example 29: *Updating the Supplier's List of Events*

```
// C++
// Find the index in the array of supplied event types
CORBA::Long NotifyPushSupplier_i::find_index(const EventType
   &e_type)
{
    for (int i = 0, i < NUM_TYPES_I_SUPPLY; i++)
    {
    if(!strcmp(e_type.domain_name,
                m_supply_types[i].domain_name) &&
       !strcmp(e_type.type_name, m_supply_types[i].type_name))
      return i;
    }

    return -1;   // Not a supplied type
}

void NotifyPushSupplier_i::subscription_change(
    const CosNotification::EventTypeSeq &added,
    const CosNotification::EventTypeSeq &removed)
throw (CORBA::SystemException)
{
    CORBA::ULong i;

1   // 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;
    }

2   // 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;
    }
}
```

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

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.

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 100) 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 `EventTypeSeq` sequence. It then invokes `offer_change()` on its structured proxy push consumer, `structured_ppc`.

```
// C++
CosNotification::EventType Baseball;
CosNotification::EventType Football;

Baseball.domain_name = CORBA::string_dup("Sports_News");
Baseball.type_name   = CORBA::string_dup("Baseball_Results");
Football.domain_name = CORBA::string_dup("Sports_News");
Football.type_name   = CORBA::string_dup("Football_Results");

CosNotification::EventTypeSeq added(2);
CosNotification::EventTypeSeq removed;
added.length(2);
added[0] = Baseball;
added[1] = Football;

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 100).

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 100).

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 30 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 30: *Subscribing to Selected Event Types*

```
// C++
// The following are defined as member variables of class
// NotifyPushConsumer_i:
CosNotification::EventTypeSeq m_consume_types;
CosNotifyFilter::FilterIDSeq m_filter_ids;

// Add a subscription for new event types chosen by user
void NotifyPushConsumer_i::add_subscription(const EventType
                                            &e_type)
{
  // Create a filter for the new subscription
  CosNotifyFilter::FilterFactory_var dff =
    channel->default_filter_factory();

  CosNotifyFilter::Filter_var filter =
    dff->create_filter("EXTENDED_TCL");
```

Example 30: *Subscribing to Selected Event Types*

```
  // Set up constraint expression for new filter
  CosNotification::EventTypeSeq event_types(1);
  event_types.length(1);
  event_types[0].domain_name =
    CORBA::string_dup(e_type.domain_name);
  event_types[0].type_name =
    CORBA::string_dup(e_type.type_name);

  CosNotifyFilter::ConstraintExpSeq constraints(1);
  constraints.length(1);
  constraints[0].event_types = event_types;
  constraints[0].constraint_expr = "";

    // Add constraint to new filter
  CosNotifyFilter::ConstraintInfoSeq *info =
    filter->add_constraints(constraint);

  m_filter_ids.length(m_filter_ids.length() + 1);
  m_consume_types.length(m_consume_types.length() + 1);

\\ Complete subscription by adding new filter to proxy
  m_filter_ids[m_filter_ids.length()-1] =
    m_proxy->add_filter(filter);

// Update internal data structures to track subscription data
  m_consume_types[m_consume_types.length()-1].domain_name =
    CORBA::string_dup(e_type.domain_name);
  m_consume_types[m_consume_types.length()-1].type_name =
    CORBA::string_dup(e_type.type_name);
}

void NotifyPushConsumer_i::init_consume_types()
{
  CosNotification::EventTypeSeq_var types_available;
1 types_available =
    m_proxy->obtain_offered_types(ALL_NOW_UPDATES_ON);

  // return with user choices
  CosNotification::EventTypeSeq_var types_wanted;
2 types_wanted = get_choices(types_available);

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

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()

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 96). A consumer's implementation should examine both arguments and re-evaluate its subscriptions accordingly.

Example

In Example 31, `offer_change()` returns new event types to an end user, who decides which (if any) of the new event types to subscribe to.

Example 31: *Adding and Removing Event Types*

```
// C++
// The following are defined as member variables of class
// NotifyPushConsumer_i:
CosNotification::EventTypeSeq m_consume_types;
CosNotifyFilter::FilterIDSeq m_filter_ids;

void NotifyPushConsumer_i::offer_change(const
   CosNotification::EventTypeSeq &added, const
   CosNotification::EventTypeSeq &removed)
throw (CORBA::SystemException)
{
1    // return with user choices
    CosNotification::EventTypeSeq_var 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

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

      if(!strcmp(removed[i].domain_name,
                 m_consume_types[n].domain_name) &&
         !strcmp(removed[i].type_name,
                 m_consume_types[n].type_name))
        {
          // Remove filter from proxy
          m_proxy->remove_filter(m_filter_ids[n]);
```

Example 31: *Adding and Removing Event Types*

```
          // Remove subscription data from customer list
          for (int ix = n; ix < m_filter_ids.length()-1; ix++)
            {
              m_filter_ids[ix] = m_filter_ids[ix + 1];
              m_consume_types[ix].domain_name =
                CORBA::string_dup(m_consume_types[ix +
                                  1].domain_name);
              m_consume_types[ix].type_name =
                CORBA::string_dup(m_consume_types[ix +
                                  1].type_name);
            } // for ix

            // Resize data structures appropriately.
            m_filter_ids.length(m_filter_ids.length()-1);
            m_consume_types.length(m_consume_types.length()-1);
          }// if !strcmp
        } // 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`.

Managing the Notification Service

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

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

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 III

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.

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

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.

Creating a Log

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*

```
// C++
#include <omg/DsNotifyLogAdmin.h>

1 CORBA::ORB_var orb = CORBA::ORB_init(argc, argv)
2 CORBA::Obect_var obj =
     orb->resolve_initial_references("NotifyLoggingService");
```

Example 32: *Obtaining a NotifyLogFactory*

```
3  DsNotifyLogAdmin::NotifyLogFactory_var factory =
      DsNotifyLogAdmin::NotifyLogFactory::_narrow(obj);
```

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

Obtain a log object

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 115 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 114. 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 115.

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 115. 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 41.

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*

```
// C++
LogId id;
```

Example 33: *Creating a NotifyLog*

```
1  // create the notification QoS properties
   QoSProperties qos;

   // create the notification Admin properties
   AdminProperties admin;

2  // set a threshold alarm at 90% full
   CapacityAlarmThresholdList thresholds;

   theresholds.length(1);
   thresholds[0] = (UShort) 90;

3  // factory obtained previously
   NotifyLog_var log = factory->create(halt, 0, thresholds, 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

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.

Logging with a BasicLog

`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*

```
// C++
1   DsLogAdmin::Anys anys;
    anys.length(3);
    anys[0] <<= "7989028321";
    anys[1] <<= "8606531000";
    anys[2] <<= "connected"

2   try
    {
      log->write_records(anys);  // log obtained previously
    }
3   catch(const DsLogAdmin::LogFull&)
    {
      cerr << "'Basic log " << log->id() << "' is full" << endl;
    }
    catch(const DsLogAdmin::LogOffDuty&)
    {
      cerr << "'Basic log " << log->id() << "' is off dutty" <<
       endl;
    }
    catch(const DsLogAdmin::LogLocked&)
    {
      cerr << "'Basic log " << log->id() << "' is locked" << endl;
    }
    catch(const DsLogAdmin::LogDisabled&)
    {
      cerr << "'Basic log " << log->id() << "’ is disabled" << endl;
    }
```

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 34 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*

```
// C++
// Create a new LogRecord
1   DsLogAdmin::LogRecord record = new DsLogAdmin::LogRecord();

// create a new attribute list
record.attr_list = new DsLogAdmin::NVList(1);
record.attr_list[0].name = CORBA::string_dup("minute_type");
record.attr_list[0].value <<= "free";

// Load the data into an any
CORBA::Any data <<= "7989028321, 8606531000, connected";
record.info = data;

// Create a RecordList
DsLogAdmin::RecordList records;
records.length(1);
records[0] = record;

2   try
    {
      log->write_recordlist(records);  // log obtained previously
    }
3   catch(const DsLogAdmin::LogFull&)
    {
      cerr << "'Basic log " << log->id() << "' is full" << endl;
    }
    catch(const DsLogAdmin::LogOffDuty&)
    {
      cerr << "'Basic log " << log->id() << "' is off duty" << endl;
    }
    catch(const DsLogAdmin::LogLocked&)
    {
      cerr << "'Basic log " << log->id() << "' is locked" << endl;
    }
    catch(const DsLogAdmin::LogDisabled&)
    {
      cerr << "'Basic log " << log->id() << "’ is disabled" << endl;
    }
```

Logging Events with an EventLog

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 CosEventChannelAdmin::SupplierAdmin_var sa =
     log->for_suppliers();
2 CosEventChannelAdmin::ProxyPushConsumer_var ppc =
     sa->obtain_push_consumer();
3 ppc->connect_push_supplier(CosEventComm::PushSupplier::_nil());
  CORBA::Any any;
  any <<= CORBA_string_dup("7989028321, 8606531000, connected");
4 ppc->push(any);
```

In step 3 a nil supplier reference is used because the log object does not need a disconnect notification.

Logging Events with a NotifyLog

`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 20.

Example

Example 37 logs events to a `NotifyLog` using a push supplier.

Example 37: *Logging events to a NotifyLog using a push supplier*

```
1  CosNotifyChannelAdmin::SupplierAdmin_var sa =
      log->default_supplier_admin();

2  CosNotifyChannelAdmin::ProxyID proxy_id;
   CosNotifyChannelAdmin::ClientType ctype =
      CosNotifyChannelAdmin::ANY_EVENT;
   CosNotifyChannelAdmin::ProxyConsumer_var obj =
      sa->obtain_notification_push_consumer(ctype, proxy_id);

3  CosNotifyChannelAdmin::ProxyPushConsumer_var pc =
      CosNotifyChannelAdmin::ProxyPushConsumer::_narrow(obj);

4  pc->connect_any_push_supplier(CosEventComm::PushSupplier::_nil(
      ));

   CORBA::Any any;
   any <<= "7989028321, 8606531000, connected";
5  pc->push(any);
```

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

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, 2014.
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 70.

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*

```
// C++
Iterator_var iterator;
RecordList_var list;

list = log->query("EXTENDED_TCL", "$minute_type == 'roaming'",
   iterator.out());
```

Deleting Records from the Log

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 124 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

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.

Scheduling

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 130 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*

```
// C++
// Contruct the times between the log is to record data
IntervalsOfDay intervals;
intervals.length(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;
sched.length(1);
sched[0].days = days;
sched[0].intervals = intervals;
```

Example 41: *Setting a logs schedule*

```
// 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`.

Log Generated Events

`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.
`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 11.

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 113.
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 31.

Example 42 implements a push consumer that receives events from the `NotifyLogFactory`.

Example 42: *Receiving events from the NotifyLogFactory*

```
// C++
1  class NotifyPushConsumer_i : public virtual
      POA_CosNotifyComm::PushConsumer
   {
     public:
   // ...

     void push(CORBA::Any event) throws (Disconected)
     {
       // Process the event
       // ...
     }
   }

   // client consumer program
   int main(int argc, char *argv[])
   {
     // ORB and POA activation not shown
     // ...

2    // Create the push consumer
     NotifyPushConsumer_i impl;
     CosNotifyComm::PushConsumer_var consumer = impl._this();

3    // get a reference to the NotifyLogFactory
     CORBA::Object_var obj =
      orb->resolve_initial_references("NotifyLoggingService");
     DsNotifyLogAdmin::NotifyLogFactory_var factory =
      DsNotifyLogAdmin::NotifyLogFactory::_narrow(obj);

     // The client consumes events of type ANY
     CosNotifyChannelAdmin::ClientType type =
      CosNotifyChannelAdmin::AnyEvent;

     // get the push proxy supplier
     CosNotifyChannelAdmin::ProxyID proxy_id;
```

Example 42: *Receiving events from the NotifyLogFactory*

```
4   try
      {
       CosNotifyChannelAdmin::ProxySupplier_var obj =
     factory->obtain_notification_push_supplier(type, proxy_id);
      }
    catch(CosNotifyChannelAdmin::AdminLimitExceeded err)
      {
      // handle the exception
      }

    CosNotifyChannelAdmin::ProxyPushSupplier_var pps =
     CosNotifyChannelAdmin::ProxyPushSupplier::_narrow(obj);

5   try
      {
      pps->connect_push_consumer(consumer)
      }
    catch (CosEventChannelAdmin::AlreadyConnected ac)
      {
      cerr << "Already connected to channel." << endl;
       exit (1);
      }
    catch (CORBA::SystemException& se)
      {
      cerr << "System exception occurred during connect: " << se
     << endl;
      exit(1);
      }

   // ...

} // main
```

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*

```
// C++
CORBA::Any_var any = // the event received by the client.

const DsLogNotification::ObjectCreation* object_creation;
const 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: *Upacking log generated events by typecode*

```
// C++
CORBA::Any_var any = // the event received by the client.

CORBA::TypeCode_var tc(any.type());
if(tc -> equivalent(DsLogNotification::_tc_ObjectCreation))
{
  // An object creation event was received.
  // Unpack the event and handle the results.
}
else if(tc ->
   equivalent(DsLogNotification::_tc_ObjectDeletion))
{
  // 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 142 and "Event Filtering" on page 59 for detailed information on event filtering.

Event Forwarding

As seen in Figure 3 on page 10 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 142 and "Log Management" on page 144.

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 142.
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 130.

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 17 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 41.

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.*

```
// C++
#include <omg/DsNotifyLogAdmin.h>
#include <omg/CosNotifyChannelAdmin.h>
#include <ong/CosNotifyComm.h>

// Implement the required supplier methods for a push supplier
class NotifyPushSupplier_i :
  public virtual POA_CosNotifyComm::StructuredPushSupplier
{
  public:
    NotifyPushSupplier_i()
    { }
    ~NotifyPushSupplier_i()
    { }

  // ...
}
```

Example 45: *Implementing the push supplier.*

```
2  // client supplier program
   int main(int argc, char *argv[])
   {
     supplier = new NotifyPushSupplier_i;

3    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv)
     CORBA::Obect_var obj =
      orb->resolve_initial_references("NotifyLoggingService");
     DsNotifyLogAdmin::NotifyLogFactory_var factory =
      DsNotifyLogAdmin::NotifyLogFactory::_narrow(obj);

4    // The log will have an id of 123
     LogId id = (ULong) 123;

     // Set the Log's QoS properties
     QoSProperties qos;
     qos.length(1);
     qos[0].name = Type;
     qos[0].value <<= QoSNone;

     AdminProperties admin;
     CapacityAlarmThresholdList thresholds;

     NotifyLog_var log = factory->create_with_id(id, halt, 0,
      thresholds, qos, admin);

5    CosNotifyChannelAdmin::SupplierAdmin_var sa =
      log->default_supplier_admin();

     CosNotifyChannelAdmin::ProxyID proxy_id;
     CosNotifyChannelAdmin::ClientType ctype =
      CosNotifyChannelAdmin::ANY_EVENT;

     try
       {
        CosNotifyChannelAdmin::ProxyConsumer_var obj =
       sa->obtain_notification_push_consumer(ctype, proxy_id);
       }
     catch(CosNotifyChannelAdmin::AdminLimitExceeded err)
       {
       // handle the exception
       }

     CosNotifyChannelAdmin::ProxyPushConsumer_ptr ppc =
      CosNotifyChannelAdmin::ProxyPushConsumer::_narrow(obj);

     try
      {
      ppc->connect_push_supplier(supplier)
      }
     catch (CosEventChannelAdmin::AlreadyConnected.value ac)
       {
       // Handle the exception
       }
```

Example 45: *Implementing the push supplier.*

```
6   CORBA::Any any;
    any <<= "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*

```
// C++
#include <omg/DsNotifyLogAdmin.h>
#include <omg/CosNotifyChannelAdmin.h>
#include <ong/CosNotifyComm.h>

1   // Implement the required methods for a push consumer
class NotifyPushConsumer_i : public virtual
   POA_CosNotifyComm::PushConsumer
{
  public:
// ...

  void push(CORBA::Any event) throws (Disconnected)
  {
   CORBA::String stock_price;
   if (!(event >>= stock_price))
     cerr << "Invalid event"  << endl
   else
     cout << "Stock price is " << stock_price << endl;
  }
}

2   // client consumer program
int main(int argc, char *argv[])
{
  consumer = new NotifyPushConsumer_i;

3   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv)
  CORBA::Obect_var obj =
   orb->resolve_initial_references("NotifyLoggingService");
  DsNotifyLogAdmin::NotifyLogFactory_var factory =
   DsNotifyLogAdmin::NotifyLogFactory::_narrow(obj);
```

Example 46: *Implementing the push consumer*

```
4   LogId id = (ULong) 123;
    NotifyLog_var log;

    if(!(log = factory->find_log(id))
      {
      cerr << "Log not found" << end;
      exit(0);
      }
5   CosNotifyChannelAdmin::ConsumerAdmin_var ca =
     log->default_consumer_admin();

    CosNotifyChannelAdmin::ProxyID proxy_id;
    CosNotifyChannelAdmin::ClientType ctype =
     CosNotifyChannelAdmin::ANY_EVENT;

    try
      {
      CosNotifyChannelAdmin::ProxySupplier_var obj =
     ca->obtain_notification_push_supplier(ctype, proxy_id);
      }
    catch(CosNotifyChannelAdmin::AdminLimitExceeded err)
      {
      // handle the exception
      }

    CosNotifyChannelAdmin::ProxyPushSupplier_ptr pps =
     CosNotifyChannelAdmin::ProxyPushSupplier::_narrow(obj);

    try
     {
     pps->connect_push_consumer(consumer)
     }
    catch (CosEventChannelAdmin::AlreadyConnected.value ac)
      {
      // Handle the exception
      }

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. Initializes the ORB and 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

`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 9 on page 142 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.

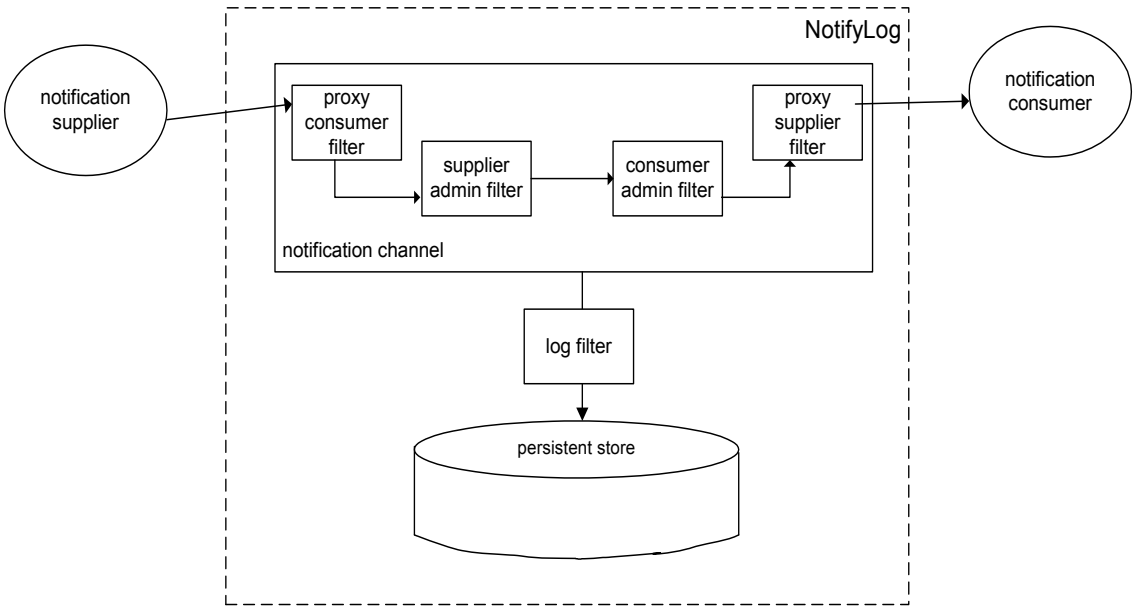

Figure 9: *Filter points in event's life-cycle*

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 59.

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 70.
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 143 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 143 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*

```
1  // C++
   // NotifyLog log obtained earlier
   CosNotifyFilter::FilterFactory_var dff =
      log->default_filter_factory();

2  CosNotifyFilter::Filter_var filter =
      dff->create_filter("EXTENDED_TCL");

3  // create a constraint
   CosNotification::EventTypeSeq event_types(1);
   event_types.length(1);

   event_types[0].domain_name = string_dup("Dial Up");
   event_types[0].type_name = string_dup("Data Error");

   CosNotifyFilter::ConstraintExpSeq constraint(1);
   constraint.length(1);

   constraint[0].event_types = event_types;
   constraint[0].constraint_expr = string_dup("$severity > '4'");

   CosNotifyFilter::ConstraintInfoSeq_var 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

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).

Administrative State

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*

```
// C++
// log obtained previously
if (log->get_administrative_state() == locked)
  {
  log->set_administrative_state(unlocked);
  cout << "Log " << log->id() << "is now unlocked." <<
   endl;
  }
```

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

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 137.

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

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

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.

Table 10: *Log QoS settings*

QoS Setting	Log Behavior
`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

Example 49 on page 149 sets a log's QoS to `QoSFlush` and then calls `flush()` on it.

Availability Status

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.

Example 49: *Setting a log's QoS properties and flushing its memory buffer*

```
// C++
//log object obtained previously
QoSList qos;
qos.length(1);
qos[0] = QoSFlush;

try
  {
  log->set_log_qos(qos);
  }
catch (DsLogAdmin::UnsupportedQoS)
  {
  // handle the exception
  }

// ...

// write the log's memory buffer to disk
log->flush();
```

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

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

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 41.

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 147.

Example 50 on page 151 sets a log's `EventReliability` and `ConnectionReliability` QoS to `Persistent`.

Example 50: *Setting notification level QoS on a NotifyLog object*

```
// C++
QoSProperties qos;
qos.length(2);
qos[0].name = EventReliability;
qos[0].value <<= Persistent;
qos[1].name = ConnectionReliability;
qos[1].value <<= Persistent;

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.

Configuring the Telecom Log Service

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 124), and the interval between flushes of the log object's internal memory buffer("Flushing the buffer" on page 148).

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 Orbix 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 147.
`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 124.
`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\6.2\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\6.2\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\6.2\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.

Glossary

A

administration
All aspects of installing, configuring, deploying, monitoring, and managing a system.

C

client
An application (process) that typically runs on a desktop and requests services from other applications that often run on different machines (known as server processes). In CORBA, a client is a program that requests services from CORBA objects.

configuration
A specific arrangement of system elements and settings.

configuration domain
Contains all the configuration information that Orbix ORBs, services and applications use. Defines a set of common configuration settings that specify available services and control ORB behavior. This information consists of configuration variables and their values. Configuration domain data can be implemented and maintained in a centralized Orbix configuration repository or as a set of files distributed among domain hosts. Configuration domains let you organize ORBs into manageable groups, thereby bringing scalability and ease of use to the largest environments. See also configuration file and configuration repository.

configuration file
A file that contains configuration information for Orbix components within a specific configuration domain. See also configuration domain.

configuration repository
A centralized store of configuration information for all Orbix components within a specific configuration domain. See also configuration domain.

configuration scope
Orbix configuration is divided into scopes. These are typically organized into a root scope and a hierarchy of nested scopes, the fully-qualified names of which map directly to ORB names. By organizing configuration properties into scopes, different settings can be provided for individual ORBs, or common settings for groups of ORB. Orbix services have their own configuration scopes.

CORBA
Common Object Request Broker Architecture. An open standard that enables objects to communicate with one another regardless of what programming language they are written in, or what operating system they run on. The CORBA specification is produced and maintained by the OMG. See also OMG.

CORBA objects
Self-contained software entities that consist of both data and the procedures to manipulate that data. Can be implemented in any programming language that CORBA supports, such as C++ and Java.

D

deployment
The process of distributing a configuration or system element into an environment.

E

event
The occurrence of a condition or state change, or the availability of some information that is of interest to one or more modules in a system. Suppliers generate events and consumers subscribe to receive them.

event channel
Accepts incoming events from client suppliers and forwards supplier-generated events to all connected consumers. From a supplier's perspective, the event channel appears as a single consumer; from a consumer's perspective, the event channel appears a a single supplier.

event service
See Orbix event service.

I

IDL
Interface Definition Language. The CORBA standard declarative language that allows a programmer to define interfaces to CORBA objects. An IDL file defines the public API that CORBA objects expose in a server application. Clients use these interfaces to access server objects across a network. IDL interfaces are independent of operating systems and programming languages.

IIOP
Internet Inter-ORB Protocol. The CORBA standard messaging protocol, defined by the OMG, for communications between ORBs and distributed applications. IIOP is defined as a protocol layer above the transport layer, TCP/IP.

installation
The placement of software on a computer. Installation does not include configuration unless a default configuration is supplied.

Interface Definition Language
See IDL.

invocation
A request issued on an already active software component.

IOR
Interoperable Object Reference. See object reference.

N

node daemon
Starts, monitors, and manages servers on a host machine. Every machine that runs a server must run a node daemon.

notification service
See Orbix notification service.

O

object reference
Uniquely identifies a local or remote object instance. Can be stored in a CORBA naming service, in a file or in a URL. The contact details that a client application uses to communicate with a CORBA object. Also known as interoperable object reference (IOR) or proxy.

OMG
Object Management Group. An open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications, including CORBA. See www.omg.com.

ORB
Object Request Broker. Manages the interaction between clients and servers, using the Internet Inter-ORB Protocol (IIOP). Enables clients to make requests and receive replies from servers in a distributed computer environment. Key component in CORBA.

Orbix event service
An implementation of the OMG Event Service Specification. Decouples communication between objects. Defines two roles for objects: a supplier role and a consumer role. Suppliers produce event data and send it to consumers through an event channel.

Orbix notification service
An implementation of the OMG Notification Service Specification. Extends the CORBA Event Service Specification to include qualities of service, subscription mechanisms, filtering and structured messages.

Orbix OTS
An implementation of the OMG Transaction Service Specification. Provides interfaces to manage the demarcation of transactions and the propagation of transaction contexts.

Orbix telecom log service
An implementation of the OMG Telecom Log Specification. The telecom log service encompasses and builds on the functionality of the event and the notification services by providing a durable and searchable log.

P

POA
Portable Object Adapter. Maps object references to their concrete implementations in a server. Creates and manages object references to all objects used by an application, manages object

state, and provides the infrastructure to support persistent objects and the portability of object implementations between different ORB products. Can be transient or persistent.

protocol
Format for the layout of messages sent over a network.

S

server
A program that provides services to clients. CORBA servers act as containers for CORBA objects, allowing clients to access those objects using IDL interfaces.

T

TCP/IP
Transmission Control Protocol/Internet Protocol. The basic suite of protocols used to connect hosts to the Internet, intranets, and extranets.

telecom log service
See Orbix telecom log service.

TLS
Transport Layer Security. An IETF open standard that is based on, and is the successor to, SSL. Provides transport-layer security for secure communications.

Index

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W