OpenFusion®
CORBA Services
Version 4.1
Notification Service

PRISMTECH

OpenFusion®
CORBA Services

NOTIFICATION SERVICE GUIDE

Part Number: OFCOR-NOTG-41

Doc Issue 19, 13 July 2004

Notices

Copyright Notice

Preface

About the Notification Service Guide

The *Notification Service Guide* is included with the OpenFusion CORBA Services' *Documentation Set*. The *Notification Service Guide* explains how to use the OpenFusion Notification Service, as well as associated extensions to the service, including the OpenFusion Typed Notification Service, OpenFusion Event Domains, and the Event Type Repository.

The *Notification Service Guide* is intended to be used with the *System Guide* and other OpenFusion CORBA Services documents included with the product distribution: refer to the *Product Guide* for a complete list of documents.

Intended Audience

The *Notification Service Guide* is intended to be used by users and developers who wish to integrate the OpenFusion CORBA Services into products which comply with OMG or J2EE standards for object services. Readers who use this guide should have a good understanding of the relevant programming languages (e.g. Java, IDL) and of the relevant underlying technologies (e.g. J2EE, CORBA).

Organisation

The *Notification Service Guide* is organised into five main sections. The first three sections describe each of the OpenFusion Notification Service components in order (Notification Service, Event Type Repository, and the Event Domain Service). Each of these sections provides

- a high level description and list of main features
- explanation of the component's architecture and concepts
- how to use specific features
- detailed explanations of the main interfaces and how to use them
- other information which is needed to use the component

The last section of the *Notification Service Guide*, *Configuration and Management* provides information on configuring and managing the OpenFusion Notification Service's components using the OpenFusion Administration Manager. Detailed descriptions of properties specific to the component are included. It is intended that this section be read in conjunction with the *System Guide*.

Conventions

The conventions listed below are used to guide and assist the reader in understanding the *Notification Service Guide*.

Item of special significance or where caution needs to be taken.

Item contains helpful hint or special information.

WIN Information applies to Windows (e.g. NT, 2000) only.

UNIX Information applies to Unix based systems (e.g. Solaris) only.

Hypertext links to WWW and other internet services are shown as *blue italic underlined.*

On-Line (PDF) versions of this document: Items shown as cross references to other parts of the document, e.g., *Contacts* on page vii, behave as hypertext links: readers can jump to that section of the document by clicking on the cross reference.

```
%  Commands or input which the user enters on the
   command line of their computer terminal
```

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown in shaded, full width boxes (to allow for standard 80 column wide text), as shown below:

```
NameComponent newName[] = new NameComponent[1];

// set id field to “example” and kind field to an empty string
newName[0] = new NameComponent (“example”, “”);

rootContext.bind (newName, demoObject);
```

Italics and ***Italic Bold*** are used to indicate new terms, or emphasise an item.

Arial Bold is used to indicate user related actions, e.g. **File | Save** from a menu.

Step 1: Indicates that this item is a step or stage of completing a task by a user.

Contacts

PrismTech can be contacted at the following address, phone number, fax and e-mail contact points for information and technical support. Users of the on-line version of this manual can *click* the e-mail addresses below to launch their e-mail client or Web browser to send e-mail direct to PrismTech.

Corporate Headquarters

PrismTech Corporation
6 Lincoln Knoll Lane
Suite 100
Burlington, MA
01803
USA

Tel: +1 781 270 1177
Fax: +1 781 238 1700

European Head Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK

Tel: +44 (0)191 497 9900
Fax: +44 (0)191 497 9901

Web: *http://www.prismtechnologies.com*

General Enquiries: *info@prismtechnologies.com*

Support Enquiries: *http://www.prismtechnologies.com/Contacts*

Contents

Table of Contents

List of Figures

List of Tables

Introduction

The *OpenFusion Notification Service* is one of a range of services and interfaces included with the *OpenFusion CORBA Services* product range.

The Notification Service component of the *OpenFusion Notification Service* product can be used stand-alone or with other OpenFusion CORBA Services interfaces and services. It is standards based, compliant with recognised industry standards and specifications, and supports portability and interoperability.

Notification Service

1 Description

1.1 Overview

The OMG Notification Service is a greatly enhanced extension of the OMG Event Service and is backwards compatible with it. Both of these services enable data, referred to as *events*, to be sent and received between distributed software objects in a *decoupled* fashion via an *event channel*. This decoupling enables events to be transmitted more efficiently and flexibly than when events are sent directly between objects (i.e., tightly coupled).

Some of the benefits of using these services include:

- ease of maintenance when adding or removing suppliers and consumers of events in a system
- more efficient use of network bandwidth between the suppliers and consumers
- performance increasingly improves over tight coupling as the number of suppliers and consumers increases (through the use of concurrency*)*

The OMG Notification Service provides additional benefits, including:

- the ability to control the flow of events in order to maximise performance
- the provision of and ability to control, event reliability within the service
- the management of the events and how their flow through the service is buffered or *queued*

The OpenFusion version 4 implementation of the Notification Service provides the majority of the features and benefits provided in the OMG Notification Service Specification, which includes those features which are most used. The OpenFusion Notification Service also provides additional benefits for improved administration of the service plus improved flexibility and control over the flow, buffering and reliability of events sent through the service.

The OpenFusion Notification Service is widely used in the telecommunications, finance, transport/travel and energy industries for applications ranging from propagating alarms on equipment, providing share dealing services, to booking hotels and planes.

OMG Standard Features

The OpenFusion Notification Service includes the standard OMG features, such as:

- decoupling the event transmission from suppliers to consumers by using *event channel*s and *proxies*. The events may be *structured* (containing details about the event), or *sequences* of events (events sent in batches for improved performance)
- avoidance of poor performance due to polling by using the *push style* event transmission model for event notification
- enabling clients to receive only those events they are specifically interested in by using filters attached to the client's proxy
- the provision of filters and the *Extended Trader Constraint Language* for controlling or limiting events being sent through the service in order to improve performance, flexibility and manageability of event transmission
- enabling reliability, e.g. guaranteed event delivery, *queues* (event flow buffers) and events to be managed at the channel, proxy or event level through the use of *Quality of Service* (QoS) settings
- facilitating the creation of filtering constraints by end-users through the provision of an *event type repository*, thus enabling information about the structure of events to be readily accessible
- enabling certain types of events to be transmitted in batches in order to increase performance
- additional administrative operations

OpenFusion Enhancements

The OpenFusion Notification Service provides many enhancements over the standard OMG specification. These enhancements include:

- provision of external graphical user interfaces, as part of the OpenFusion Graphical Tools, for run-time administration of the service
- rich administrative interface
- an extensive Quality of Service framework incorporating additional settings for improved controlability, performance tuning and flexibility
- provisions for improved performance and scalability, such as
  - multi-threading
  - ability to *federate* channels (connect event channels together)
  - provision of persistence for events, channels and connections to commercial databases through the use of optimized stored procedures
  - automatic service activation on demand

- support for custom Java filters which may perform substantially better than the standard OMG constraint filter
- ability to federate channels across multiple platforms and interoperate with native notification services

1.2 Concepts and Architecture

Although the OpenFusion Notification Service is generally compliant with the OMG Notification Service specification, it has many additional features and enhancements.

The OMG Notification Service is an extension of the OMG Event Service and is backwards compatible with it. However, this release of the OpenFusion Notification Service only supports the semantics specified for Notification Service clients, since a vast majority of users only use this client type.

Dependencies on Other Services

The Notification Service does not require other services in order to run. However, the Notification Service IDL includes IDL from these services:

- Notification Service inherits from the Event Service.
- Time Service definitions are used to support start time and timeout values.

The Time Service can be used to provide a central source of time within a distributed system when a client wishes to time-stamp events. The Timer Event Service can be used to generate events at timed intervals.

The Basic Concept

There are many situations when an object needs to receive notification that an event has been generated or produced by another object, such as when an alarm control panel of a security system needs to know if a remote alarm has been activated. The object may also need to know details about the event itself so that it can take appropriate action. Using the security system example, the alarm panel may need to know which alarm was activated, its location, the reason for the alarm (break-in, fire, etc.) in order to provide appropriate information to security officers.

Obviously, the objects producing and using the event need to be connected to each other in some fashion so that communication of the event can occur. A simple solution would be to connect the objects together directly: notification of an event occurrence and information about it being communicated *directly* between the two

objects. Importantly, these objects would then be *tightly coupled* to each other: changes effecting the communication of the event by one object will directly affect the other object.

Tight coupling performs well when one object is connected to only one other object. If, however, many objects are connected to many others, especially when the number of objects changes, then maintainability, performance and scalability become serious issues. For example, each time an event *producer* object (e.g. a new alarm) is added, then all event user, or *consumer*, objects (e.g. the alarm panels in the building, at the security firm, in the police or fire stations) will need to be changed, too. In software terms, code for all consumer objects, i.e. the *consumers*, will need to be altered, re-compiled, tested, etc., whenever supplier objects, i.e. the *suppliers*, are added.

Also, communication between tightly coupled objects is *synchronous*, that is before the supplier can send an event, the consumer must be ready to receive it. If a supplier is connected to several consumers, then it must wait for the slowest consumer to receive (or consume) the event before it can proceed.

Decoupling suppliers and consumers through an intermediary can overcome these issues. If new suppliers or consumers are added to the system, then only the intermediary needs to be altered, not each consumer or supplier, respectively. Further, the intermediary can provide event buffers, or *queues* and *multi-threading* capabilities in order to enable *asynchronous* communication: events can be sent and received without waiting for the slowest “member of the pack”.

An intermediary can therefore take over the task of communicating events between suppliers and consumers: it can provide a *service* for them, who become its *clients*.

The Event Service was the first service that the OMG specified for the decoupled, asynchronous communication of events between event producer and consumer client objects. By decoupling the objects, through the use of an *event channel* and *proxies*, the Event Service provided improved maintainability, performance and scalability over systems which rely on tightly coupled objects.

Like the Event Service, the Notification Service provides decoupled, asynchronous communication between supplier and consumer client objects. However, the Notification Service provides additional features, such as *Quality of Service* and *filtering*, to dramatically improve reliability and help control event transmission.

The Architecture

The Notification Service can be looked at from two perspectives:

1. from the journey that an event takes from supplier to consumer, i.e. its transmission path

2. how the Notification Service components are conceptually connected and created

Event Transmission

A supplier generates events.

1. The supplier sends the events to a proxy representing the consumer, the *consumer proxy*. If needed, the event can be *translated* to a type that is expected by the consumer.
2. Unwanted events can be filtered out before transmission to the next stage of the journey, the *supplier admin object*.
3. Numerous consumer proxies can be connected to a single supplier admin object: filtering and quality of service settings can be applied by the admin object to all of the events being supplied by the proxies, as a group, before they are sent to the *event channel*.
4. The event channel transmits the events, which have not been filtered out, to a *consumer admin object*. The consumer admin object then forwards those events to its individual *supplier proxies*: additional filtering and quality of service adjustments can be by defined the admin object prior to forwarding.
5. Each supplier proxy sends their events to their respective event consumers (one proxy per consumer). Final filtering and quality of service settings can be applied at the proxy for each event before it is sent on to the consumer.

Figure 1, Basic OpenFusion v.4 Implementation, shows that only the *push* model of event transmission is used in the OpenFusion v.4 implementation of the basic architecture.

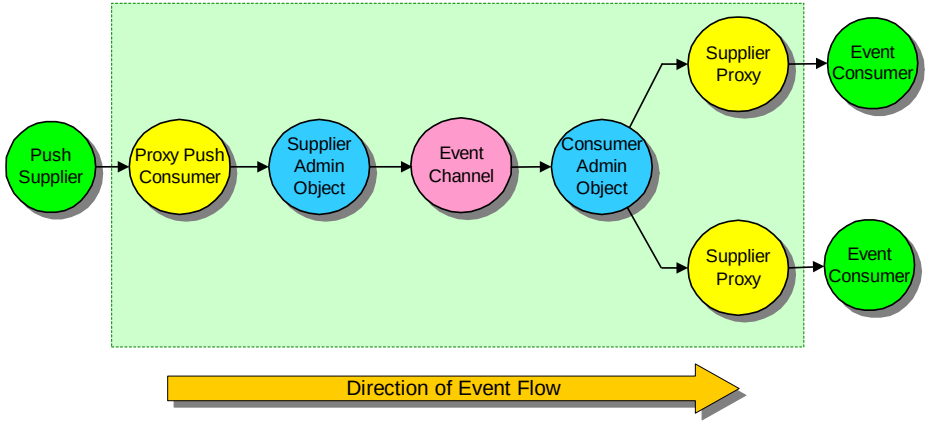

Figure 1 Basic OpenFusion v.4 Implementation

Component Connection and Creation

The components of the service are organised hierarchically. The main component is the event channel. Event channels are created by the service's event channel factory: multiple event channels can be created by the event channel factory for operation within the service.

Admin objects are created by event channels; proxies are created from the admin objects. Finally, each proxy is connected to a client supplier object or client consumer object.

Each object within the hierarchy is given a unique identifier when it is created. The combination of the hierarchical organisation and the unique identifiers enables all components to be found or referenced from any other component in the hierarchy.

Main Components and Features

The main components of the OpenFusion Notification Service are:

- event channels, admin objects, proxies, filter objects, queues and an Event Type Repository

The types of event are:

- structured events (the OpenFusion Notification Service does ***not*** support *Event Style* events or *typed* events, although they may be supported in future releases)

The transmission model is the push model (note that the OMG-defined pull model is rarely used and was removed from the OpenFusion Notification Service in order to reduce size and complexity and improve performance).

Figure 2, Main Components, shows the service's main components, including filters, queues, translation and the Event Type Repository.

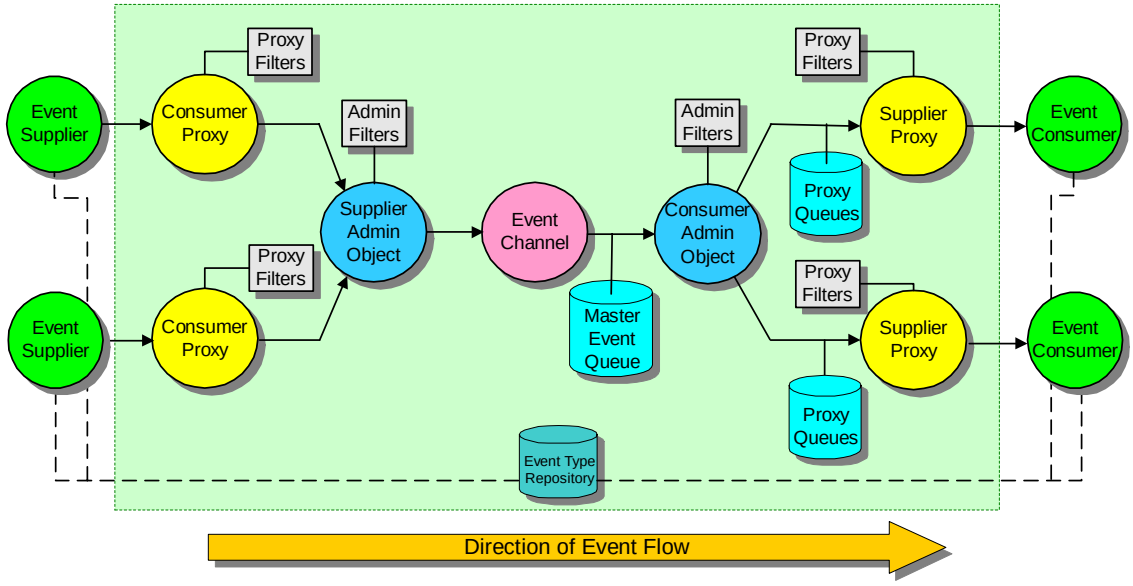

Figure 2 Main Components

Features

The Notification Service provides various management, reliability and performance enhancing operations and features, including:

- standard OMG features
  - *Quality of Service* (QoS), for providing and controlling reliability, queue management and event management
  - *sequencing*, enabling events to be sent in batches in order to enhance performance
- OpenFusion enhancements:
  - *Quality of Service extensions*, additional QoS properties for improving controllability and flexibility of event transmission
  - *federation*, where event channels can be connected or federated together for performance, reliability and flexibility
  - *transparent fail-over*, which takes advantage of ORB vendor features (when provided) for keeping the service operating when a server host fails; enables another host to transparently, without loss of events, support the service
  - *persistence*, which enables events and connections to be made persistent

- *event storage plugins*, enables database storage of persistent events, including the use of JDBC and stored procedures
- *administration tools*, including Graphical User Interfaces (as part of the OpenFusion product) and additional programming interfaces (as part of the service itself)

These components, event types, transmission models, methods and features will be described in detail below.

The Details

Structured Events

Untyped events encapsulate basic data types transmitted and received by client objects. Structured events are untyped events with attached headers containing id, QoS and filtering information.

A structured event consists of two main parts:

- an *event header* containing identification and Quality of Service information and
- an *event body* containing information used to filter the event, plus the event itself, an Any

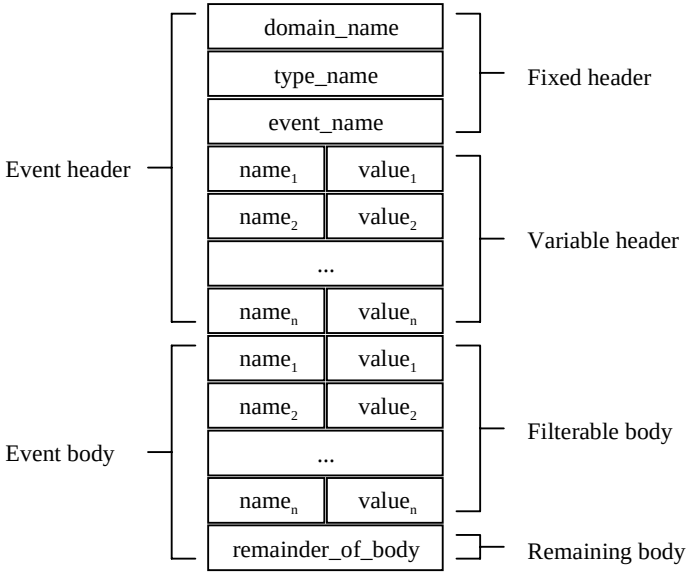

Figure 3 Structured Event

Event Header

The event header contains a *fixed header* and *variable header.*

The fixed header holds information identifying the particular event and includes:

- an *event domain* (`domain_name`) - the domain of a particular vertical industry where the event type is defined, such as *telecommunications*, *finance*, *transportation*, etc.
- an *event type* (`type_name`) - the type of particular event within the domain, for example *StockQuote* within the *finance* domain
- an *event name* (`event_name`) - a unique name for the particular event instance being transmitted

The event domain and event type can be used in combination as an index into the Event Type Repository (see *Event Type Repository* below).

The variable header contains QoS property settings for a specific event. These settings consist of a sequence of zero or more *name-value* pairs. The name component of the pair is a string variable which identifies a particular QoS property; the value component is an `Any` which contains the value of the QoS property.

For example, a name could be set to the QoS property *EventReliability* with its corresponding value set to 1 (a `short` defined as *persistent*). Refer to *Quality of Service* on page 21 for a list of available QoS properties.

Event Body

The event body contains a *filterable body* and a *remaining body.*

The filterable body contains another sequence of zero or more name-value pairs. These pairs, predictably, are used for filtering the event. Each name-value pair consists of the name of a property (a `string` variable) and its value (an `Any`).

The filterable body is intended to be used for filterable properties which have been defined within an application domain. In order to filter the event, a client constructs filter constraints which are applied, using the Notification Service's filters, to the properties contained in the structured event's filterable body. (See *Filtering* on page 22.)

The remaining body (`remainder_of_body`) contains the actual event data, which is an `Any`. As with the original Event Service, this part of the structured event can contain any data that a user wants to send along with the event.

Event Type Repository

The OMG specifies that the Event Type Repository is an optional feature of the Notification Service; this feature is provided in the OpenFusion implementation of the service.

The Event Type Repository is a facility for making it easier for clients to create event filters by making information about the structure of events available to clients.

The Event Type Repository stores information about the kinds of filterable data that specified events can provide to consumers. The repository only contains information about the properties contained in the filterable body of a structured event (see *Structured Events* on page 14).

The repository can be queried by event suppliers to discover the names and types of the properties that an event of a certain type contains. The supplier can use this information to send events which conform to that type.

The repository can also be used by event consumers in order to determine which properties are expected by events of a certain type; the consumer must create the expression to match the event they are interested in.

Importantly, the Event Type Repository has the ability to modify event types and the relationship between event types at run time. This allows applications to evolve over time. For example, an application can create a new event type, with additional properties, that inherits from an existing event type. New applications can take advantage of the additional information, while existing applications can process the event according to the old set of properties.

Event Communication Models

The OpenFusion Notification Service uses the *push* communication model, whereby suppliers actively send or push events to the event channel and consumers passively receive them

Event Channel

The event channel (also referred to as the *notification channel* in the Notification Service) is the component which provides the loosely-coupled communication between client objects. It is the event channel which handles supplier registration and the broadcasting of events to consumers.

The Notification Service allows any number of event channels to be active concurrently.

Notification Service event channels, unlike those of the Event Service, possess Quality of Service (QoS) properties and event filtering. QoS and filters set on a channel affect all relevant events which pass through it. Further, QoS and filter settings are inherited by any admin object created by the event channel.

Client objects can set various QoS and administrative properties on the event channel when it is created. For example, some of the properties that can be set include the maximum number of events the channel will buffer at a time, as well as the maximum number of consumers and suppliers that can connect to the channel.

Event channels are created by an *event channel factory*. The channels, in turn, create admin objects, which in turn create proxies. This creation process forms a channel - admin - proxy hierarchy.

Note that when a new channel is created, indeed when any object in the hierarchy is created, it is given a unique numeric identifier. This identifier enables objects within the hierarchy to find (i.e. find a reference to) their ‘parent’ or ‘child’ objects. This ability enables objects to administer other objects within their hierarchy. Clients are therefore able to discover all objects that comprise the hierarchy, starting from any object within the channel.

Admin Objects

Admin objects perform various administrative and management functions, such as creating proxies and acting as a mechanism for separating proxies into controllable groups

Admin objects are associated with either suppliers or consumers (supplier admin objects or consumer admin objects).

i Note that *supplier* admin objects create *consumer* proxies and vice versa (remembering that suppliers connect to consumer proxies, consumers connect to supplier proxies). The Notification Service’s admin objects can create, in addition to Notification Style proxies, Event Service style proxies.

Event channels may have multiple admin objects. This enables proxies to be logically grouped and to optimise the handling of clients which have identical requirements.

Admin objects manage or administer the proxies that they have created (as a group):

- QoS properties are assigned to an admin object’s proxies at the time the proxy is created, although the QoS properties for these proxies can be changed for each individual proxy as required

- an admin object's filter properties (by assigning a *filter object* to it) affect all the proxies connected to it, even though each proxy may have its own, additional filter objects

Proxies

Proxies connect supplier and consumer client objects to the event channel of the Notification Service. Importantly, proxies represent or stand-in for a client. For example, a supplier behaves as if it is connected to an actual consumer, however it is actually connected to a proxy for the consumer, i.e. a *consumer proxy*[1]: suppliers connect to consumer proxies; consumers connect to supplier proxies.

Individual proxy types are specific to:

- the type of event being transmitted
- whether the events are being sent singly or in batches when used with structured events (referred to as *sequenced structured events*)

For example, a *structured push supplier proxy* connects a *structured event consumer* to the event channel and uses the *push model* to receive events.

Each proxy has its own QoS object plus zero or more filter objects: this enables QoS properties and filter properties to be set at the individual proxy level. Note, however, that the QoS and filter object settings for the proxy's admin object also affect the events that the proxy receives or transmits. For example, a proxy consumer (connected to a suppler) may allow *Event A* to be sent, but its admin object may still filter it out.

Suspension, Resumption and Disconnection

Push-model event suppliers can temporarily suspend event communication. The event channel buffers the events while a consumer connection is suspended: these events are transmitted when the client resumes its connection (subject to the QoS discard policy when the maximum number of events per consumer QoS policy is exceeded).

Figure 4 illustrates the four states a proxy can have during creation, suspension, resumption and disconnection.

1. Also called *proxy consumer*: both forms are used in the OMG specification

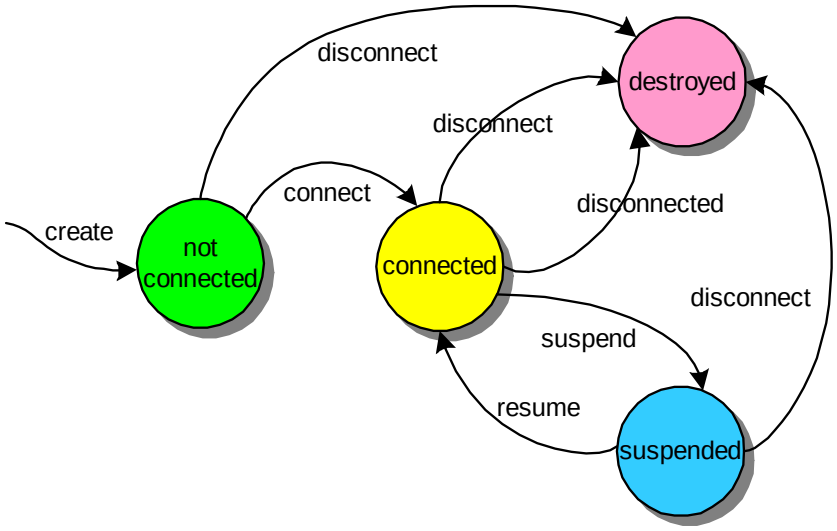

Figure 4 Proxy States

For proxy push suppliers, the suspended state indicates that the Notification Service will suspend the pushing of events onto the consumer. While suspended, events will be queued at the proxy for later delivery.

A proxy is a communication end point and disconnecting it implies that the proxy object is destroyed. After being disconnected, the proxy can no longer be used to send or receive events.

A push consumer can also disconnect a proxy by raising the `Disconnected` exception in the push operation.

It is the client's responsibility to disconnect (and destroy) the proxy when the client terminates since the service has no means of knowing that the client no longer exists. Accordingly, the client should call its associated proxy's disconnect method. For example, if the client is a push supplier connected to a *`ProxyPushConsumer`* (suppliers connect to consumer proxies, consumers connect to supplier proxies), then the *`disconnect_push_consumer()`* method for its `ProxyPushConsumer` object should be called prior to termination.

Queues

Queues are buffers for storing events until consumers are ready to receive the events. Queues free suppliers from the need to wait for consumers to consume their events before continuing.

Each event channel has a master event queue and each supplier proxy has a proxy queue, one proxy queue per consumer object (see *Figure 5*).

Incoming events enter the master event queue: if event reliability is set to persistent, the event will be written to persistent storage before the event is sent on. The behaviour of the master event queue is affected by the event channel's *order* and *discard* QoS policies. The queue's maximum length is set by the `MaxQueueLength` property.

Events are then dispatched into proxy queues. Each proxy queue has its own *order* and *discard* policies for the proxy object it is connected to, i.e. each proxy queue may have different policies than the others. The maximum queue size for a proxy queue is limited by the `MaxEventsPerConsumer` QoS property.

The proxy queues potentially contain very different sets of events, depending on filtering, ordering, queue size and the "speed" of the consumer. When an event is delivered, it is removed from the master queue.

The proxy queue keeps track of the events which have been delivered. If the Notification Service fails for any reason (e.g. host crash, lost connection, etc.), then the contents of the master queue will be recovered, provided that the events have been set as persistent beforehand. Note that when recovery takes place only those events which have not yet been delivered to a consumer will be allowed to re-enter the proxy queue.

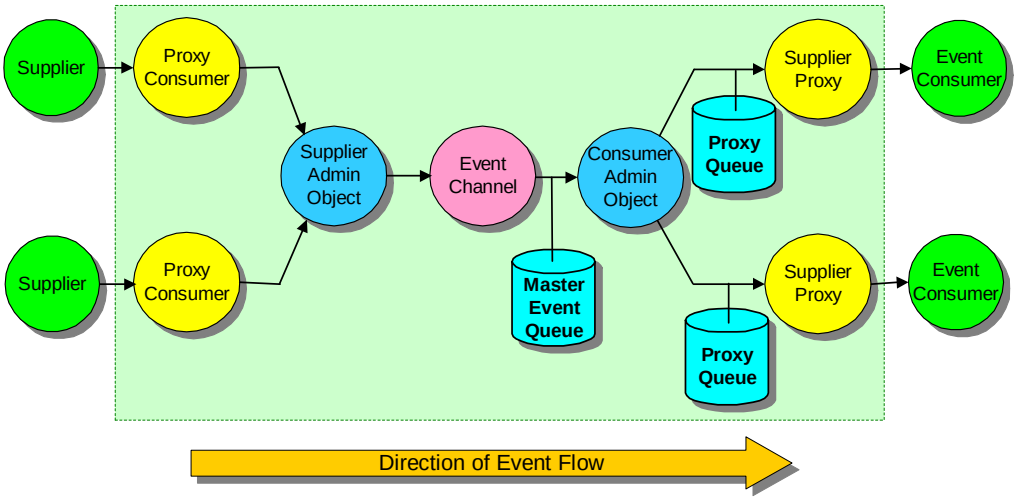

Figure 5 Event Queues

OpenFusion Queue Extensions

The OpenFusion implementation of the Notification Service provides a number of queue management extensions in addition to the standard OMG interfaces. These additional extensions can be used to access useful information or functionality that is not provided otherwise.

Quality of Service

There is no direct communication between suppliers and consumers when using the Notification Service (a decoupled communication model). Consequently, when an event is sent from a supplier to a consumer, there are three points where the event is (conceptually) transmitted:

1. when the event is delivered by the supplier to the event channel
2. when it is forwarded by the channel
3. when the event is delivered by the channel to the consumer

An application may wish to set QoS at each of these points. Accordingly, the Notification Service enables each channel, connection and message (the transmission points) to possess relevant, configurable QoS settings. These settings cover the delivery guarantee, aging characteristics and prioritisation for the transmitted events.

Standard OMG Properties

Quality of Service settings are defined as *properties*; each property has an associated *value*. A particular property may have a range of values that indicate different requirements or delivery characteristics to support a wide variety of application needs: precise QoS requirements, at any particular level, can be expressed as a *set* of properties.

Quality of Service properties cover three main areas: reliability, queue management and event management. Note that not all QoS properties can be applied at all levels of the Notification Service.

Detailed descriptions of these properties are given under *Supplemental Information, Standard OMG Properties.* on page 69.

OpenFusion QoS Extensions

The OpenFusion Notification Service supports the QoS properties described in the OMG specification which are listed above. Further, the OpenFusion Notification Service supports a comprehensive, extensible QoS framework that allows clients to configure the run-time behaviour of event channels, admin and proxy objects: in other words, their QoS properties can be set at run-time.

The OpenFusion Notification Service's QoS also:

- enforces portability, especially with regard to reliability
- supports ORB vendor features
- addresses the Event Service's deficiencies
- provides additional queuing policies

The extended OpenFusion Notification Service QoS properties are listed and described under *Supplemental Information*, *OpenFusion QoS Extensions* on page 73.

The QoS framework supports logical *grouping*, whereby a channel treats its admin objects as a group and an admin object treats its proxies as a group.

A group is a collection of objects that have been created by a particular factory, the *group object*. For example, a channel, the group object (or *group* for short) groups the admin objects it has created; an admin object is the group object for its proxies.

The value of a QoS property that has been applied to a group automatically becomes the default value for all new objects created by that group. Note that existing objects, those previously created by the group object, are not affected. Also note that a client may override existing QoS group properties for any object within the group.

Filtering

Filtering allows the transmission of events to be selectively stopped or filtered out. Filtering is performed using *filter objects* which are attached to admin and proxy objects (see *Figure 2, Main Components* on page 13). A single filter object can be added to more than one of these objects at a time: for example a single filter can be used by several proxies, or by a proxy and an admin. However, this can lead to unmanageable deployment situations (see warning note shown immediately below).

Filter objects should be destroyed when the objects that use them are destroyed, otherwise they will become a source of leakage. However, care must be taken when destroying filter objects that are used by multiple objects in order to avoid inadvertently destroying a filter which is still in use.

Filter objects use a *constraint language* to describe which events should be filtered, i.e. they constrain which events are allowed and may be referred to as *forward filters* since they forward filtered events. Also, all constraints added to a filter are assigned a unique identifier which enables constraints to be modified or deleted at run-time.

Constraint Language

Any conformant implementation of the Notification Service specification must support the *Extended Trader Constraint Language* (Extended TCL), an extension of the constraint language used for the Trading Service.

The Extended TCL grammar fixes a few problems with the basic Trader Constraint Language, while adding suitable constructs for filtering events.

This grammar is intuitive for programmers because it mimics how data structures are normally accessed and is based on the Java style *dot* notation.

For example, a simple query string could be:

```
$type_name == 'Alarm' and $Priority > 4
```

which forwards events of type `Alarm` which have a priority greater than four.

A description of the Extended TCL grammar and how to use filter constraints with the Notification Service is given under *Writing Constraint Expressions* on page 56.

Sequencing

The Notification Service supports the transmission of *sequences of Structured Events* (*event sequencing* for short). Event sequencing is a process or technique whereby one or more events are transmitted at a time as a single IIOP package. Event sequencing boosts the event transmission performance of the service: sending an IIOP package with one event and sending an IIOP package with 100 events takes approximately the same amount of time.

There are separate *sequence* clients and proxies which are used for transmitting sequences of Structured Events (see *Figure 6*).

Event sequencing uses the `MaximumBatchSize` and `PacingInterval` QoS properties. These properties can only be applied on the consumer side:

- `MaximumBatchSize` - The maximum number of events that a consumer wishes to receive at a time. Consumers should always set this QoS since the default value is one.
- `PacingInterval` - The maximum time the consumer is willing to wait for the batch to fill. At the end of the pacing interface, the Notification Service will deliver whatever events it has. The default value is zero (indefinite wait).

The Notification Service will wait at least until one event is available before delivering any events to the consumer. If no events are available, the Notification Service will therefore wait longer than the pacing interval.

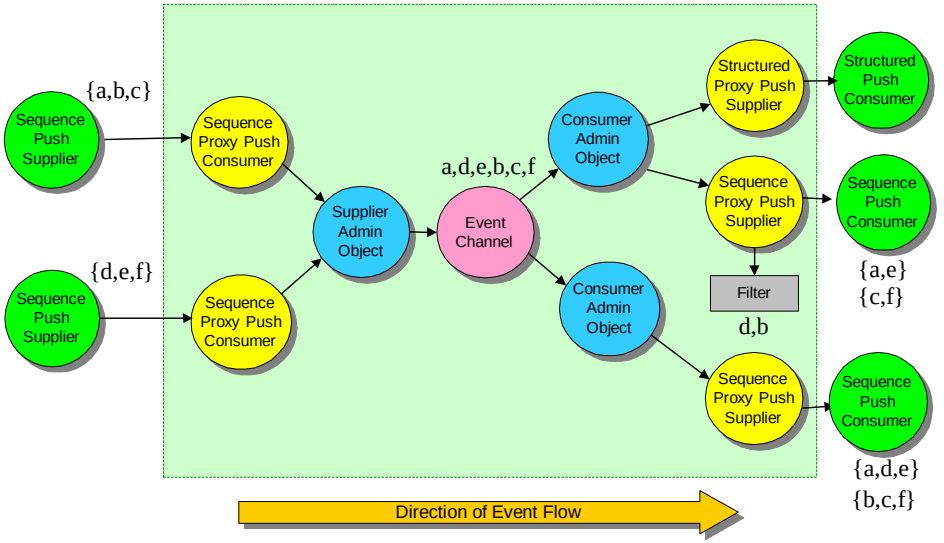

Figure 6 Sequencing Architecture

All events delivered by all connected suppliers will be included in the event sequences arriving on the consumer side.

i Event sequencing does not influence the order of events transmitted through the channel (notice the order of the events as received by the consumers in *Figure 6*). However, ordering can be controlled by using QoS properties and filters.

Auto-sequencing

Auto-sequencing provides a significant performance improvement for structured proxies without changing how the proxies function externally. When auto-sequencing is used, a proxy uses internal batching to send multiple structured events in one CORBA call: this provides the performance increase usually associated with a sequence proxy. Externally, however, a structured proxy push supplier still sends structured events individually to the consumer and a structured proxy push consumer still receives structured events individually from the supplier.

Auto-sequence functionality is used exclusively by *structured* proxies, not by the *sequence* proxies described in the previous section.

There are characteristics of auto-sequencing which make it unsuitable for some situations:

- A failure of the service can result in a loss of a number of events up to the maximum batch size.

- If a supplier process terminates (by invoking `System.exit()` or returning from its `main()` method, for example), events up to the maximum batch size may be lost. To avoid this situation in a controlled shutdown, suppliers shuld call `disconnect()` before the process ends. This will cause any pending events to be delivered to the channel.
- Exceptions cannot be sent back to a caller. For example, a structured proxy push supplier will not be able to report to the event channel when it has failed to push events onto a structured consumer.

Auto-sequencing should not be used if persistence or error detection are important issues.

Two QoS properties, `AutoSequenceBatchSize` and `AutoSequenceTimeout`, are used to control auto-sequence functionality.

By default, auto-sequence functionality is switched **on** in an OpenFusion installation. If it is not required, it should be switched off using the appropriate QoS settings (as described on page 75).

Persistence

The OpenFusion implementation of the Notification Service provides the ability to make events and connections persistent.

The OpenFusion Framework and by association the OpenFusion Notification Service, provides the facility to add components as *plugin modules* for supporting different application requirements. The event persistence is enabled and managed through:

- *event database plugins* which connect the service to a selected database, such as Oracle and
- additional QoS properties which are provided in the Notification Service

Features

The persistence feature of the OpenFusion Notification Service provides improved reliability by enabling the use of a recovery strategy

Requirements

There are a number of factors to be aware of when using persistence:

- event reliability can only be set to persistent if the connection reliability is also set to persistent
- the client must be a persistent CORBA object

- its proxy must only be connected once
- the proxy is disconnected when the `OBJECT_NOT_EXIST` ORB system exception is thrown
- the proxy must be suspended when the client object is *passivated*
- QoS properties must be set for:
  - maximum queue size(s)
  - reconnect interval

A persistent client is a persistent CORBA object. A persistent object can be activated and passivated several times, but in terms of the ORB (and thus the Notification Service) it is the same object.

When a server with persistent client objects is re-started (or the object is otherwise activated), the client must not create a new proxy since it will continue to use the proxy that was used prior to passivation.

The Notification Service will retry persistent clients until it encounters an `OBJECT_NOT_EXIST` system exception. This exception is normally raised when the object is de-registered from the BOA or POA.

Persistent clients should use a number of QoS properties to control resources. The discard policy and maximum queue size should be used for consumers to limit the number of events that are queued on their behalf.

The reconnect interval can be set to reduce the frequency at which the Notification Service retries an unavailable object.

Push consumers can also suspend these proxies prior to passivation in order to avoid interaction while the object is unavailable.

Passivating Persistent Clients

Persistent clients are automatically re-connected when they re-register with the ORB. A persistent client would normally save the proxy `IOR` when it connects to the Notification Service the first time.

When a persistent client is passivated, the ORB will raise standard `NO_IMPLEMENT` system exceptions when the Notification Service attempts to deliver or retrieve events, or do event type callbacks.

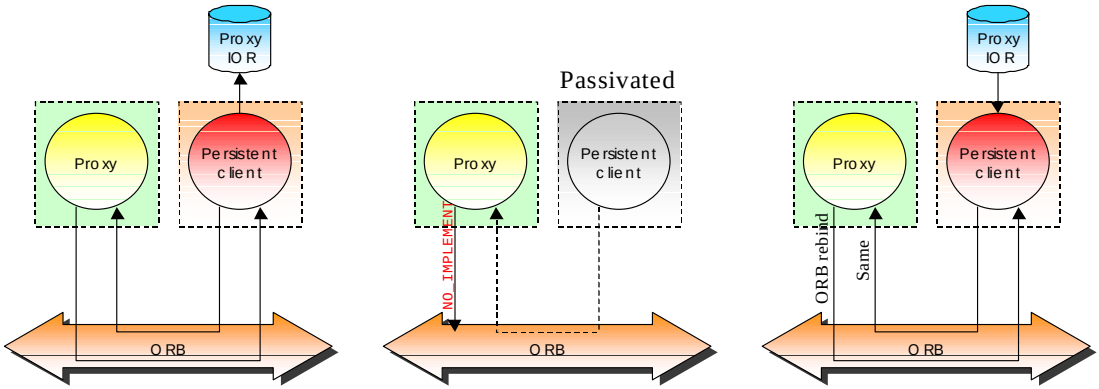

Figure 7 Passivating Persistent Clients

When the persistent client is later activated, the ORB will rebind the connection between the Notification Service and the client. This happens automatically and the client should not connect to a new proxy.

The client normally loads the proxy `IOR` from file or, for example, the Naming Service upon restart. The proxy is needed for later connection manipulation (suspend, resume), filter administration and ultimately disconnecting.

If a client de-registers from the ORB, the ORB will raise an `OBJECT_NOT_EXIST` exception when the Notification Service tries to interact with the client. This will disconnect the client.

Federation

Federation is a method of connecting separate Notification Service instances and their event channels together (see *Figure 8, Federation of Channels Architecture*).

Federation effectively creates a composite system partitioned into any number of subsystems. Partitioning an event system into multiple “event subsystems” can have a number of advantages:

- Performance:
  - enabling multiple hosts to be used for utilising increased CPU resources
  - providing fan-out to consumers on the local machine

    Sending events to a channel that in turn forwards them to a number of consumers can result in great performance improvements. As an example, if the consumers are all on the same machine the events can be sent using one network invocation and a series of local invocations.

- Reliability:

  - avoiding single points of failure

    By having multiple event channels it is possible to avoid single points of failure. Although parts of the system may no longer receive events if an event channel fails, this does not necessarily have to affect other consumers.

- Flexibility:

  - makes it easy to move event subsystems

  - can use filtering to control fan-in and fan-out

    Grouping suppliers and consumers into logical units can simplify system configuration and improve flexibility. For instance, instead of changing all consumers in a group to use a new channel, only the suppliers that provide events to the group would need to be altered.

Referring to *Figure 8*, the fact that a consumer proxy is a supplier and a proxy supplier is a consumer allows channels to be federated without using special clients that forward events from one channel to another. The inheritance structure described allows a proxy supplier to be connected directly to a proxy consumer.

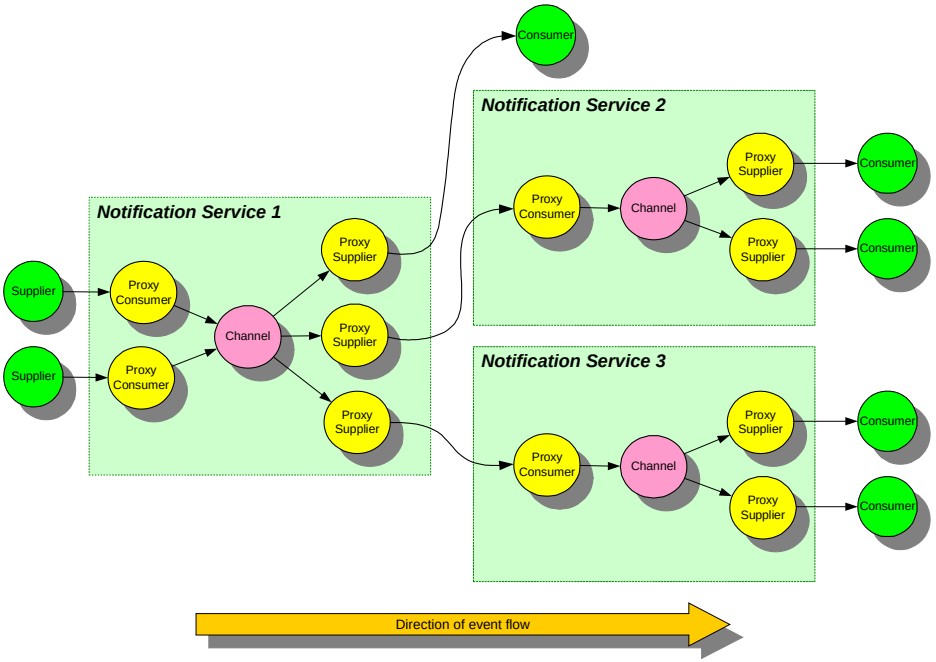

Figure 8 Federation of Channels Architecture

Local Channel

The local channel concept (*Figure 9*) provides failure support for dumb clients which assume that the Notification Service is always available.

Local channel protection is only intended to recover from node failures and not process failures.

Suppliers and consumers may always create a proxy, connect and just start sending or receiving events: connection reliability would be set to *best effort* on the client side of the channel.

The federation connections would be persistent to ensure they are re-established after a node crash. It is possible to use a separate Notification Service as the intermediator, or use direct connections.

Referring to *Figure 9*, if Host C becomes unavailable, the proxy supplier on Host A (or Host B) will queue all incoming events until the receiving Notification Service becomes available again.

In order to be certain that the consumer doesn't lose events, it may be necessary to make the consumer persistent. This would avoid a situation where the proxy consumer starts receiving persistent events before the consumer has connected.

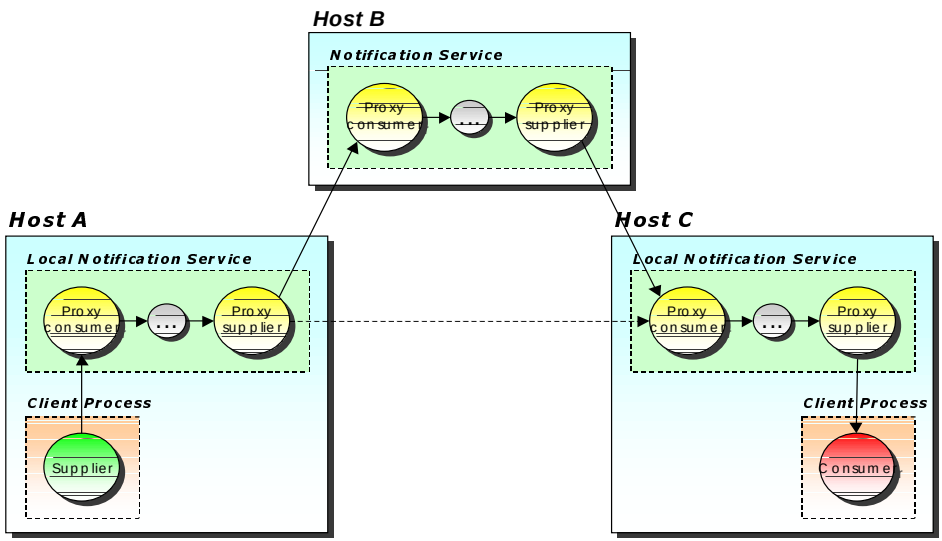

Figure 9 Local Host

2 Using the Service

2.1 Introduction

The main tasks which are performed when using the Notification Service include:

- initialising the ORB and the Notification Service
- creating event suppliers, which requires
  - connecting to the Notification Service event channel
  - creating events
  - sending events
- creating event consumers, which requires
  - connecting to the Notification Service event channel
  - receiving events
- setting QoS properties
- creating and applying event filters

This section describes how the specific features of the Notification Service can be used to achieve the tasks listed above. The section is organised into a sequence of topics which

- give general instructions for compiling and running Notification Service clients
- describe basic aspects of creating Notification Service clients
- describe advanced features of Notification Service clients such as QoS and event filtering

Each topic uses examples to illustrate how the tasks can be accomplished. Additional examples, complete with source code and descriptions of how to compile and run them, are supplied separately as part of the OpenFusion product distribution.

Note

- All of the example code used in this section requires that the OpenFusion Notification Service is installed and running.
- There is little or no error-checking in the examples shown here. Code to deal with exceptions has generally been omitted for the sake of clarity and brevity. These exceptions must of course be properly caught and handled in a working system.

Import Statements

The following packages are required to be imported into classes which are Notification Service clients. This list is not exhaustive: additional packages may be required depending on the specific features of the client.

Standard Notification Service Features

The following packages support OMG standard Notification Service features

```
org.omg.CosNotification.*
org.omg.CosNotifyComm.*
org.omg.CosNotifyFilter.*
org.omg.CosNotifyChannelAdmin.*
org.omg.CosTypedNotifyComm.*
org.omg.CosTypedNotifyChannelAdmin.*
```

OpenFusion Extensions

The following package is needed when using the OpenFusion Notification Service extensions:

```
com.prismt.cos.CosNotification.NotificationExtensions.*
```

2.2 Compiling and Running Clients

This section describes the general principles to follow when compiling and running Notification Service clients.

Compiling Client Applications

Clients written for the OpenFusion Notification Service must be compiled with a supported Java compiler. See the OpenFusion release notes for supported Java versions.

For further instructions, consult the documentation supplied with your Java compiler. The are no specific compiler options needed in order to compile Notification Service clients.

Running Client Applications

Before running any Notification Service client applications, the Notification Service must be running on one of the supported ORBs.

Initialising the ORB

The appropriate ORB daemon should be running before the Notification Service is started. Full instructions for how to run your ORB will be given in your ORB documentation. For example, when running JacORB use the following command:

```
%  imr
```

The *OpenFusion Product Guide* lists supported ORBs and their start-up/run commands.

Starting the Notification Service

Step 1: Ensure your PATH contains the `bin` directory of the JDK and the `bin` directory of the OpenFusion distribution. The UNIX scripts (or Windows `.bat` files) that start the Notification Service are located in the `bin` directory.

Step 2: Ensure the appropriate ORB daemon is running (see above).

Step 3: Start the Notification Service from a command prompt using the following command:

```
%  server -start NotificationService
```

The same command can be used at either a UNIX or Windows command prompt.

Alternatively, start the OpenFusion Administration Manager and use the GUI tools to start and configure the Notification Service. The *System Guide* gives details of using the Administration Manager and other options for running OpenFusion services.

Configuring the Notification Service

The OpenFusion Notification Service can be installed and run “out of the box” with no additional configuration. It is strongly recommended, however, that you configure the service to optimise performance and reliability for your specific environment. Section 13, *Notification Service Configuration*, on page 171 describes every configurable service property. All properties can be set programatically, or see the *System Guide* for details of how to set properties through the GUI Administration Manager.

All of the example code given in this section can be run using the default (out of the box) Notification Service configuration.

Starting Clients

Once the Notification service is running and suitably configured, client applications can be started.

i The Notification Service must be running *before* any clients are started, otherwise clients will be unable to create or resolve event channels and thus unable to function.

Also note that in most cases consumers should be started *before* suppliers are started, otherwise events may be lost as suppliers begin pushing them onto the event channel before there is a consumer available to receive them.

2.3 Creating Clients

Notification Service clients include both suppliers and consumers. This section provides a simple example of each, showing how the key features that every client must possess can be implemented. Advanced client features, such as filtering and setting QoS, are covered in subsequent sections.

Creating a Supplier

The first task a Notification Service supplier must perform is to locate the Notification Service server instance and connect to it. Connections are made to an event channel, via proxy and admin objects.

Connecting to the Server

Step 1: Obtain an object reference to the event channel factory.

Event channels are created by the Notification Service's event channel factory. Before an event channel can be created, an object reference to the factory must be obtained. The ORB's `resolve_initial_references` method is passed the name `NotificationService` and this is used to resolve initial references to locate the object

```
org.omg.CORBA.Object object = null;
org.omg.CORBA.ORB orb = null;

try
{
   object = orb.resolve_initial_references (“NotificationService”);
}
catch (org.omg.CORBA.ORBPackage.InvalidName ex)
{
   System.err.println (“Failed to resolve Notification Service”);
   System.exit (1);
}
```

At this point, the type of the object referenced by `object` is an undefined of type `org.omg.CORBA.Object`. The `narrow` method of the `EventChannelFactoryHelper` helper class is used to narrow the returned object reference to a specific `EventChannelFactory` object.

```
EventChannelFactory factory = null;

factory = EventChannelFactoryHelper.narrow (object);
```

Step 2: Create an event channel or obtain a reference to an existing channel.

New event channels can be created once the reference to the factory has been obtained (step 1). The example below uses the `factory` object's `create_channel` method to create a new channel with default Quality of Service settings.

```
Property[] qos = new Property[0];
Property[] adm = new Property[0];
org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder ();
EventChannel channel = null;

try
{
   channel = factory.create_channel (qos, adm, id);
}
catch (UnsupportedQoS ex) {}
catch (UnsupportedAdmin ex) {}
```

Further details of setting QoS properties when the channel is created are given in *Creating an Event Channel with QoS* on page 48.

Managing Event Channels

Once the event channel has been created, the supplier may need to perform other actions upon it. To this end, the following example shows how the supplier might obtain a reference to a specific event channel.

First, the `get_all_channels` operation returns a sequence of channel identifiers:

```
int ids[] = factory.get_all_channels ();
```

Next, the `get_event_channel` operation is used to obtain an `EventChannel` object from an identifier:

```
Vector vector = new Vector ();

for (int i = 0; i < ids.length; i++)
{
   try
   {
      vector.addElement (factory.get_event_channel (ids[i]));
   }
   catch (ChannelNotFound ex) {} // ignore
}

EventChannel all[] = new EventChannel [vector.size ()];
for (int i = 0; i < all.length; i++)
```

```
{
   all[i] = (EventChannel) vector.elementAt (i);
}
```

The event channel objects are collected in a vector in order to account for the situation when other interactions are happening with the event channel factory at the same time. This strategy illustrates general practice when dealing with distributed systems.

Destroying an Event Channel

The supplier might also be responsible for destroying the event channel once it is no longer needed.

Event channels are destroyed using the `destroy` operation:

```
channel.destroy ();
```

All administration objects and all proxy objects created by the administration objects are destroyed along with the channel. Also, all suppliers and consumers connected to this channel are disconnected and any events which have yet to be delivered are discarded. Note that the object reference to a channel is invalidated when it is destroyed

Step 3: Get the `SupplierAdmin` object reference.

Supplier administration objects in the Notification Service are created using the `new_for_suppliers` operation. This operation takes a filter operator *`in`* parameter and a unique identifier *`out`* parameter and returns a newly created administration object:

```
InterFilterGroupOperator sop = InterFilterGroupOperator.AND_OP;
org.omg.CORBA.IntHolder sid = new org.omg.CORBA.IntHolder ();
SupplierAdmin sadm = channel.new_for_suppliers (sop, sid);
```

The `InterFilterGroupOperator` object specifies how filters attached to an administration object are combined with filters attached to the proxies created by the administration object. The Notification Service supports the following settings for the filter operator:

- *`AND`*: Both an administration filter and a proxy filter must pass an event in order for the event to be forwarded.
- *`OR`*: The event is forwarded when either an administration filter or a proxy filter passes an event.

Managing Administration Objects

Administration objects are managed via an array in a similar manner to the event channels described in Step 2. The following code shows how to create a list of all `SupplierAdmin` objects in an event channel:

```
int ids[] = channel.get_all_supplieradmins ();
Vector vector = new Vector ();

for (int i = 0; i < ids.length; i++)
{
   try
   {
      vector.addElement (channel.get_supplieradmin (ids[i]));
   }
   catch (AdminNotFound ex) {} // ignore
}

SupplierAdmin all[] = new SupplierAdmin [vector.size ()];
for (int i = 0; i < all.length; i++)
{
   all[i] = (SupplierAdmin) vector.elementAt (i);
}
```

Step 4: Obtain a structured push consumer proxy object.

The supplier admin object supports operations for creating proxy consumers. In the example code below, the SupplierAdmin object *`admin`*, obtained in Step 3, is used to produce proxy consumers (in other words, proxies which represent consumers). The example shows the creation of three types of consumer.

First, create holders which will hold the IDs of the proxies for each of the three types:

```
org.omg.CORBA.IntHolder anyID = new org.omg.CORBA.IntHolder ();
org.omg.CORBA.IntHolder strID = new org.omg.CORBA.IntHolder ();
org.omg.CORBA.IntHolder seqID = new org.omg.CORBA.IntHolder ();
```

The client types which will be used are then specified and saved to *`ClientType`* variables:

```
ClientType anyType = ClientType.ANY_EVENT;
ClientType strType = ClientType.STRUCTURED_EVENT;
ClientType seqType = ClientType.SEQUENCE_EVENT;
```

The `ProxyPushConsumer` variables for each of the three types are declared. This is followed by the declaration of three `ProxyConsumer` variables:

```
ProxyPushConsumer anyProxy;
StructuredProxyPushConsumer strProxy;
SequenceProxyPushConsumer seqProxy;

ProxyConsumer pc1 = null;
ProxyConsumer pc2 = null;
ProxyConsumer pc3 = null;
```

The supplier admin object's `obtain_notification_push_consumer` method is called to obtain a reference to the correct proxy object. For each proxy, the *identity* and *type* parameters are passed. The return for this call is always a `ProxyConsumer`:

```
try
{
   pc1 = admin.obtain_notification_push_consumer (anyType, anyID);
   pc2 = admin.obtain_notification_push_consumer (strType, strID);
   pc3 = admin.obtain_notification_push_consumer (seqType, seqID);
}
catch (AdminLimitExceeded ex)
{
   System.err.println ("Admin limit exceeded!");
   System.exit (1);
}
```

The final stage uses helper classes to cast the objects into their correctly typed proxies:

```
anyProxy = ProxyPushConsumerHelper.narrow (pc1);
strProxy = StructuredProxyPushConsumerHelper.narrow (pc2);
seqProxy = SequenceProxyPushConsumerHelper.narrow (pc3);
```

Managing Proxies

The administration interfaces support a number of operations for managing the created proxies. The following code:

1. Obtains the unique identifier, the channel and the filter operation
2. Lists the total number of proxies
3. Examines whether or not the proxy with identifier `42` exists for a `SupplierAdmin` object called `admin`

```
int[] pushProxies = admin.push_consumers ();

int total = pushProxies.length;

System.out.println ("Total proxies: " + total);

try
{
   ProxyConsumer proxy = admin.get_proxy_consumer (42);
   System.out.println ("Proxy with id 42 exists!");
}
catch (ProxyNotFound ex)
{
   System.out.println ("Proxy with id 42 doesn't exist!");
}
```

Step 5: Connect to the proxy.

To connect to a proxy use the `connect_structured_push_supplier` method.

In the following code, `strProxy` is the reference to the structured push consumer proxy obtained in step 4. The `connect_structured_push_supplier` method is used to connect a structured push supplier object to it.

```
try
{
   strProxy.connect_structured_push_supplier
      (StructuredPushSupplierHelper.narrow
         (ObjectAdapter.getObject (this)));
}
catch (org.omg.CosEventChannelAdmin.AlreadyConnected ex)
{
   System.err.println (“Already connected!”);
   // Handle exception
   return;
}
```

Step 6: Disconnect from the proxy.

To disconnect the supplier from the proxy consumer, use the `disconnect_structured_push_consumer` method:

```
strProxy.disconnect_structured_push_consumer ();
```

The proxy object is invalidated and cannot be used when it has been disconnected.

i Further options for proxy management can be found in *Removing Inactive Proxies* on page 47.

Creating Events

Structured events consist of header and body components. The header consists of properties added to the event as an array. The body consists of data in the form of a CORBA Any. These components are created using the methods illustrated in the following example:

```
StructuredEvent event = new StructuredEvent ();

Property variable[] = new Property [2];
variable[0] = new Property ();
variable[0].name = Priority.value;
variable[0].value = orb.create_any ();
variable[0].value.insert_short ((short) 4);
variable[1] = new Property ();
variable[1].name = Timeout.value;
variable[1].value = orb.create_any ();
variable[1].value.insert_ulonglong ((long) 4*10*1000*1000); // 4 seconds

Property filterable[] = new Property [2];
filterable[0] = new Property ();
filterable[0].name = “packets”;
filterable[0].value = orb.create_any ();
filterable[0].value.insert_long (2000);
filterable[1] = new Property ();
filterable[1].name = “username”;
filterable[1].value = orb.create_any ();
filterable[1].value.insert_string (“client 1”);
```

```
EventType type = new EventType (“Telecom”, “Info”);
FixedEventHeader fixed = new FixedEventHeader (type, “event”);

org.omg.CORBA.Any data = orb.create_any ();
data.insert_long (42);

event.header = new EventHeader (fixed, variable);
event.filterable_data = filterable;
event.remainder_of_body = data;
```

This example creates a structured event with the following components:

- QoS settings `priority` (short) and `timeout` (unsigned long) in the variable header
- filterable properties `packets` (long) and `username` (string) in the filterable body
- domain name `Telecom` (string)
- type name `Info` (string)
- some data (long)

Sending Events

Events in the Notification Service are transmitted by client objects implementing one of the following Supplier interfaces:

- *`PushSupplier`*
- *`StructuredPushSupplier`*
- *`SequencePushSupplier`*

A supplier can begin sending events as soon as it has obtained a proxy of the corresponding type and has connected to it. The event supplier typically obtains its events from some external source or produces events when some external event has occurred. See *Creating Events* on page 39 for an example of how to create a structured event.

A typical event supplier must perform each of the steps listed below.

Step 1: Resolve an event channel factory. Code for this is given in *Connecting to the Server*, step 1 on page 34.

Step 2: Obtain a reference to an event channel. Code for this is given in *Connecting to the Server*, step 2 on page 35.

Step 3: Obtain a reference to a supplier admin object. Code for this is given in *Connecting to the Server*, step 3 on page 36.

Step 4: Obtain a reference to a proxy consumer object. Code for this is given in *Connecting to the Server*, step 4 on page 37.

Step 5: Connect to the proxy consumer. Code for this operation is given in *Connecting to the Server*, step 5 on page 38.

Step 6: After the supplier has established a connection to the proxy consumer, it can begin pushing events onto the event channel.

The following code uses an infinite loop to send a continuous stream of simple events. (This is suitable for test purposes; in reality, events would normally be sent when created by some triggering mechanism.)

```
while (true)
{
   org.omg.CORBA.Any data = orb.create_any ();
   obtain_data (data); // obtain data from external source

   StructuredEvent event = new StructuredEvent ();

   EventType etype = new EventType (“example”, “test”);
   FixedEventHeader fixed = new FixedEventHeader (etype, “event”);

   Property variable[] = new Property[0];

   event.header = new EventHeader (fixed, variable);
   event.filterable_data = new Property[0];
   event.remainder_of_body = data;

   try
   {
      proxy.push_structured_event (event);
   }
   catch (org.omg.CosEventComm.Disconnected ex) {}
}
```

In this example, the data of the structured event is obtained by invoking the `obtain_data` method, which gets the data from an external source. The proxy’s `push_structured_event` method is used to push the event onto the event channel.

Creating a Consumer

The first task a Notification Service consumer must perform is locate the Notification Service and connect to it. Connections are made to an event channel, via proxy and admin objects.

Connecting to the Server

Step 1: Obtain an object reference to the event channel factory. The method is identical to that used in suppliers, as described in *Creating a Supplier* on page 34:

```
org.omg.CORBA.Object object = null;
org.omg.CORBA.ORB orb = null;
```

```
try
{
   object = orb.resolve_initial_references (“NotificationService”);
}
catch (org.omg.CORBA.ORBPackage.InvalidName ex)
{
   System.err.println (“Failed to resolve Notification Service”);
   System.exit (1);
}
```

```
EventChannelFactory factory = null;

factory = EventChannelFactoryHelper.narrow (object);
```

Step 2: Create an event channel or obtain a reference to an existing channel. The method is identical to that used in suppliers, as described in *Creating a Supplier* on page 34:

```
org.omg.CORBA.IntHolder cid = new org.omg.CORBA.IntHolder ();
Property[] qos = new Property[0];
Property[] adm = new Property[0];
EventChannel channel = null;
try
{
    channel = factory.create_channel (qos, adm, cid);
}
catch (UnsupportedQoS ex) {}
catch (UnsupportedAdmin ex) {}
```

Step 3: Get the `ConsumerAdmin` object reference.

Consumer administration objects in the Notification Service are created using the `new_for_consumers` operation. This operation takes a filter operator *`in`* parameter and a unique identifier *`out3...==`* parameter and returns a newly created administration object:

```
InterFilterGroupOperator cop = InterFilterGroupOperator.AND_OP;

org.omg.CORBA.IntHolder cid = new org.omg.CORBA.IntHolder ();

ConsumerAdmin cadm = channel.new_for_consumers (cop, cid);
```

The `InterFilterGroupOperator` object specifies how filters attached to an administration object are combined with filters attached to the proxies created by the administration object. The Notification Service supports the following settings for the filter operator:

- *`AND`*: Both an administration filter and a proxy filter must pass an event in order for the event to be forwarded.
- *`OR`*: The event is forwarded when either an administration filter or a proxy filter passes an event.

Managing Administration Objects

Administration objects are managed via an array in the same manner as suppliers manage admin objects. The following code shows how to create a list of all `ConsumerAdmin` objects in an event channel:

```
int ids[] = channel.get_all_consumeradmins ();
Vector vector = new Vector ();

for (int i = 0; i < ids.length; i++)
{
   try
   {
      vector.addElement (channel.get_consumeradmin (ids[i]));
   }
   catch (AdminNotFound ex) {} // ignore
}

ConsumerAdmin all[] = new ConsumerAdmin [vector.size ()];
for (int i = 0; i < all.length; i++)
{
   all[i] = (ConsumerAdmin) vector.elementAt (i);
}
```

Step 4: Obtain a structured push supplier proxy object.

The consumer admin object supports operations for creating proxy suppliers. In the example code below, the ConsumerAdmin object *`admin`*, obtained in step 3, is used to produce proxy suppliers (in other words, proxies which represent suppliers). The example shows the creation of three types of supplier.

First, create holders which will hold the IDs of the proxies for each of the three types:

```
org.omg.CORBA.IntHolder anyID = new org.omg.CORBA.IntHolder ();
org.omg.CORBA.IntHolder strID = new org.omg.CORBA.IntHolder ();
org.omg.CORBA.IntHolder seqID = new org.omg.CORBA.IntHolder ();
```

The client types which will be used are then specified and saved to *`ClientType`* variables:

```
ClientType anyType = ClientType.ANY_EVENT;
ClientType strType = ClientType.STRUCTURED_EVENT;
ClientType seqType = ClientType.SEQUENCE_EVENT;
```

The `ProxyPushSupplier` variables for each of the three types are declared. This is followed by the declaration of three `ProxySupplier` variables:

```
ProxyPushSupplier anyProxy;
StructuredProxyPushSupplier strProxy;
SequenceProxyPushSupplier seqProxy;

ProxySupplier ps1 = null;
ProxySupplier ps2 = null;
ProxySupplier ps3 = null;
```

To initially obtain a reference to the correct proxy object, the call `obtain_notification_push_supplier` is made on the consumer admin object. For each proxy, the parameters for *identity* and *type* are passed. The return for this call is always a `ProxySupplier`:

```
try
{
   ps1 = admin.obtain_notification_push_supplier (anyType, anyID);
   ps2 = admin.obtain_notification_push_supplier (strType, strID);
   ps3 = admin.obtain_notification_push_supplier (seqType, seqID);
}
catch (AdminLimitExceeded ex)
{
   System.err.println (“Admin limit exceeded!”);
   System.exit (1);
}
```

The final stage uses helper classes to cast the objects into their correctly typed proxies:

```
anyProxy = ProxyPushSupplierHelper.narrow (ps1);
strProxy = StructuredProxyPushSupplierHelper.narrow (ps2);
seqProxy = SequenceProxyPushSupplierHelper.narrow (ps3);
```

Managing Proxies

The administration interfaces support a number of operations for managing the created proxies. The following code:

1. Obtains the unique identifier, the channel and the filter operation.
2. Lists the total number of proxies.
3. Examines whether or not the proxy with identifier `42` exists for a `ConsumerAdmin` object called `admin`.

```
int[] pushProxies = admin.push_suppliers ();

int total = pushProxies.length;

System.out.println (“Total proxies: “ + total);

try
{
   ProxySupplier proxy = admin.get_proxy_supplier (42);
   System.out.println (“Proxy with id 42 exists!”);
}
catch (ProxyNotFound ex)
{
   System.out.println (“Proxy with id 42 doesn’t exist!”);
}
```

Step 5: Connect to the proxy.

Use the `connect_structured_push_consumer` method to connect to a proxy.

In the following code, *proxy* is the reference to structured push consumer proxy obtained in Step 4. The `connect_structured_push_consumer` method is used to connect a structured push consumer object to it.

```
try
{
   strProxy.connect_structured_push_consumer
      (StructuredPushConsumerHelper.narrow
                (ObjectAdapter.getObject (this)));
}
catch (org.omg.CosEventChannelAdmin.AlreadyConnected ex)
{
   System.err.println (“Already connected!”);
   // Handle exception
   return;
}
catch (org.omg.CosEventChannelAdmin.TypeError ex)
{
   System.err.println (“Type error!”);
   // Handle exception
   return;
}
```

Step 6: Disconnect from the proxy.

To disconnect the consumer from the proxy supplier, use the `disconnect_structured_push_supplier` method, as follows:

```
strProxy.disconnect_structured_push_supplier ();
```

The proxy object is invalidated and cannot be used when it has been disconnected.

i Further options for proxy management can be found in *Removing Inactive Proxies* on page 47.

Receiving Events

Events in the Notification Service can be received by client objects implementing one of the following Consumer interfaces.

- *PushConsumer*
- *StructuredPushConsumer*
- *SequencePushConsumer*

Push consumers receive events by implementing a push operation that corresponds to the consumer type. Note that responsive push consumers should return from the push operation as quickly as possible. One way to achieve this would be to provide event processing within a separate thread.

The following code shows a simple implementation of the push operation used by structured push consumers:

```
public void push_structured_event (StructuredEvent event)
```

```
{
   org.omg.CORBA.Any data = event.remainder_of_body;
   int value = data.extract_long ();
   System.out.println (“Received event: “ + value);
}
```

The `extract_long` method extracts the data from the incoming event. In this example, we assume that the data is an integer value. If the supplier had formed the event in a different way, putting a string in the event body, for example, a different extraction method would be required.

Suspending and Resuming Connections

Event consumers of the push type can temporarily suspend event communication. To prevent event loss when a consumer connection is suspended, the event channel buffers the events sent by the supplier. When the connection is re-established, event transmission to the consumer resumes with potentially no loss of events.

In practice, the event loss on reconnection is controlled by Quality of Service properties. The `MaxEventsPerConsumer` QoS property determines how many events will be held for a disconnected consumer. See Section 4.1, *Quality of Service Properties* on page 69 for a description of the `MaxEventsPerConsumer` property.

To suspend a connection, the client should call the proxy’s `suspend_connection` operation as shown in the following example:

```
try
{
   strProxy.suspend_connection ();
}
catch (ConnectionAlreadyInactive ex)
{
   System.err.println (“Already suspended!”);
   // handle exception
}
catch (NotConnected ex)
{
   System.err.println (“Not connected!”);
   // handle exception
}
```

To resume a suspended connection, the client should call the proxy’s `resume_connection` method as shown in the following example:

```
try
{
   strProxy.resume_connection ();
}
catch (ConnectionAlreadyActive ex)
{
   System.err.println (“Already resumed!”);
   // handle exception
}
catch (NotConnected ex)
{
   System.err.println (“Not connected!”);
   // handle exception
```

```
}
```

Removing Inactive Proxies

A common requirement in the Notification Service is to remove inactive supplier and consumer proxies when they are no longer needed (because they are connected to suppliers or consumers that no longer exist).

This section gives guidance on how this is handled for different types of proxy.

Proxy Push Consumers and Proxy Pull Suppliers

When the proxy has been idle for a specified period of time, the proxy is disconnected. The amount of idle time required before disconnection should be specified with the `MaxInactivityInterval` Quality of Service property, described on page 75.

Proxy Push Suppliers

The way that proxy push suppliers are handled depends on the setting of the `ConnectionReliability` Quality of Service property.

With Connection Reliability set to Best Effort

If the `ConnectionReliability` QoS on the proxy is set to `BestEffort`, the Notification Service will always destroy a proxy push supplier when it fails to deliver an event to its attached consumer.

With Connection Reliability set to Persistent

If the `ConnectionReliability` QoS is set to `Persistent`, the Notification Service will keep resending events until an `OBJECT_NOT_EXIST` system exception is encountered. The conditions that raise this exception are ORB-specific. Most ORBs raise the exception only when the object no longer exists; in this case, the proxy can be safely removed. The following ORBs throw `OBJECT_NOT_EXIST` correctly:

- VisiBroker 3.4
- VisiBroker 5.0
- OrbixWeb 3.2
- Orbix 2000 v1.2
- Orbix 2000 v2.0
- JacORB 1.3
- JacORB 1.4

However, a number of ORBs raise the exception if the object is merely inactive, in which case it is not always safe to remove the proxy. The following ORBs have this behaviour:

- VisiBroker 4.1
- VisiBroker 4.5
- Orbacus 4.0
- Orbacus 4.1

When `OBJECT_NOT_EXIST` cannot be used reliably, the `MaxReconnectAttempts` and `ReconnectInterval` QoS properties can be used. `MaxReconnectAttempts` defines the maximum number of times the Notification Service will attempt to reconnect to a failed pull supplier or push consumer. The Notification Service disconnects the client (as though the disconnect operation had been invoked on the proxy) if the client is still unavailable after the maximum number of attempts have been made. `ReconnectInterval` determines the interval the Notification Service will wait between reconnect attempts.

Alternative Method

To determine whether a given proxies (of any type) is inactive, the `ConnectedClient` QoS property can be used. This property is set on all proxies and gives the object reference of the connected client. Use `get_qos()` on the proxy to obtain the property array and loop through the array to locate the `ConnectedClient` property (see *Accessing the QoS* on page 50 for an example of this). The value of the `ConnectedClient` property contains the object reference of the client associated with that proxy. From this, it is possible to determine if the client exists and whether the proxy can therefore be safely destroyed.

2.4 Using Quality of Service Properties

Quality of Service settings may be applied to event channels, admin objects and proxy objects on either the supplier or the consumer side. The following example demonstrates how to apply QoS to an event channel.

Creating an Event Channel with QoS

QoS properties and administrative properties are applied to an event channel when it is created by passing an array of properties as a parameter of the `create_channel` operation. The following example illustrates this. The example code given here can be part of a either a supplier or a consumer.

Step 1: Create an array to hold the QoS properties. In this example, the array is sized to hold two properties.

```
Property[] qos = new Property[2];
```

Step 2: Add the QoS properties to the array. Each array element holds a property name and a property value. The following code adds the `EventReliability` property to the array and sets its value to *persistent*.

```
qos[0] = new Property ();
qos[0].name = EventReliability.value;
qos[0].value = orb.create_any ();
qos[0].value.insert_short (Persistent.value);
```

Similarly, the following code adds the `ConnectionReliability` property to the array and sets its value to *persistent*.

```
qos[1] = new Property ();
qos[1].name = ConnectionReliability.value;
qos[1].value = orb.create_any ();
qos[1].value.insert_short (Persistent.value);
```

Step 3: Repeat the above steps to create an array of administrative properties. Although the procedure is the same as for QoS properties, a separate array is required as the `create_channel` method takes two separate array parameters. The following code creates an array of one element and populates it with the `MaxQueueLength` property, setting the property's value to *100*.

```
Property[] adm = new Property[1];
adm[0] = new Property ();
adm[0].name = MaxQueueLength.value;
adm[0].value = orb.create_any ();
adm[0].value.insert_long (100);
```

Step 4: Use the event channel factory's `create_channel` operation to create the channel, passing the Qos and administrative property arrays as parameters, as illustrated by the following code:

```
org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder ();
EventChannel channel = null;

try
{
   channel = factory.create_channel (qos, adm, id);
}
catch (UnsupportedQoS ex) {}
catch (UnsupportedAdmin ex) {}
```

i The Notification Service throws exceptions with detailed information when the code attempts to set illegal QoS or administrative properties.

Managing QoS

QoS and administrative properties do not have to be set when the event channel is created. Properties can be altered programatically at any time and new properties can be added to the channel.

Adding New QoS to a Channel

Adding a new QoS or administrative property to an existing channel requires the channel's `set_qos` or `set_admin` operations. These operations take an array of properties as a parameter. The array of properties is constructed exactly as in *Creating an Event Channel with QoS* on page 48.

The following code illustrates how to use `set_qos` to add the `MaximumBatchSize` QoS property:

```
Property newQoS[] = new Property[1];

newQoS[0] = new Property ();
newQoS[0].name = MaximumBatchSize.value;
newQoS[0].value = orb.create_any ();
newQoS[0].value.insert_long (100);

try
{
   channel.set_qos (newQoS);
}
catch (UnsupportedQoS ex) {}
```

The following code illustrates how to use set_admin to add the `MaxQueueLength` administrative property:

```
Property newAdm[] = new Property[1];
newAdm[0] = new Property ();
newAdm[0].name = MaxQueueLength.value;
newAdm[0].value = orb.create_any ();
newAdm[0].value.insert_long (10);

try
{
   channel.set_admin (newAdm);
}
catch (UnsupportedAdmin ex) {}
```

Accessing the QoS

The QoS and administrative settings for a channel can be accessed using the channel's `get_qos` and `get_admin` operations. The following code illustrates a way of simply listing the current value of each property:

```
Property qos[] = channel.get_qos ();
Property adm[] = channel.get_admin ();

for (int i = 0; i < qosP.length; i++)
{
   System.out.println ("Name : " + qos[i].name);
}
```

```
for (int i = 0; i < admP.length; i++)
{
   System.out.println (“Name : “ + adm[i].name);
}
```

Validating Event QoS

Supplier and consumer proxies provide an operation for validating the QoS setting of an event. The operation is `validate_event_qos` and is defined in the `ProxyConsumer` and `ProxySupplier` interfaces.

It is good practice for all suppliers that use QoS settings in the header of a structured event to use this operation to validate the settings before sending an event.

```
Property[] qos = new Property[2];
NamedPropertyRangeSeqHolder available;

qos[0] = new Property ();
qos[0].name = Priority.value;
qos[0].value = orb.create_any ();
qos[0].value.insert_short ((short) 4);
qos[1] = new Property ();
qos[1].name = Timeout.value;
qos[1].value = orb.create_any ();
qos[1].value.insert_ulonglong ((long) 4*10*1000*1000); // 4 seconds

available = new NamedPropertyRangeSeqHolder ();

try
{
   proxy.validate_event_qos (qos, available);
}
catch (UnsupportedQoS ex)
{
   System.err.println (“Unsupported QoS settings!”);
   // Handle exception.
}
```

2.5 Using Filters

Filters can be attached to both admin objects and proxies on both the supplier and the consumer side. Filters that are attached to admin objects apply to all the proxies created by that admin object.

An object with attached filters will only forward an event when one or more of the filters passes the event.

Filter Objects

Filters are objects in their own right and must be treated as distinct from the admin or proxy objects they are attached to. An individual filter object can be used by more than one admin or proxy object.

There are two important points to keep in mind when managing filters:

- A filter exists independently of the proxies that is associated with: if an associated proxy is destroyed or the proxy's reference to the filter is removed, then the filter will still exist. Accordingly, it is recommended that the filter's reference is stored so that it can still be referenced or destroyed after its associated proxies are removed.
- A filter should be destroyed only ***after*** all proxies referencing the filter have removed their references to it, otherwise the proxies may contain hanging references (which may subsequently throw an exception).

Take care to avoid leaving references to non-existent filters or creating orphaned filter objects which have no references to them.

Creating a Filter Object

The recommended way to create a filter is by using the event channel's filter factory, as this creates the filter in the same process as the admin and proxy objects which will use it.

Step 1: Obtain a reference to a filter factory by invoking the channel's `default_filter_factory` object, as in the following code:

```
FilterFactory filterFactory = channel.default_filter_factory ();
```

Step 2: Use the factory's `create_filter` operation to create the filter object.

The `create_filter` operation takes the name of the filter grammar as a parameter. Currently, the only grammar supported by the Notification Service is Extended TCL, so the string `EXTENDED_TCL` must be passed to the `create_filter` operation. The following code illustrates this.

```
Filter filter = null;
String grammar = "EXTENDED_TCL";

try
{
   filter = filterFactory.create_filter (grammar);
}
catch (InvalidGrammar ex)
{
   System.err.println ("Grammar " + grammar + " is invalid!");
   // Handle exception
}
```

Adding a Filter Object to an Admin Object

Use the admin object's `add_filter` operation to add a filter to the object, as follows:

```
int id = admin.add_filter (filter);
```

Listing Filter Objects

The following example shows how to obtain a list of filters attached to an admin object and then use that list to perform management operations on each item in the list (in this case, to verify that the correct filter grammar is being used).

```
int[] all = admin.get_all_filters ();
Vector vector = new Vector ();

for (int i = 0; i < all.length; i++)
{
   try
   {
      Filter f = admin.get_filter (all[i]);
      vector.addElement (f);
   }
   catch (FilterNotFound ex) {}
}

for (int i = 0; i < vector.size(); i++)
{
   Filter f = (Filter) vector.elementAt (i);
   if (! f.constraint_grammar().equals (“EXTENDED_TCL”))
   {
      System.err.println (“Filter has unknown grammar!”);
      // Handle exception
   }
}
```

Removing Filter Objects

To remove a single, specified filter from an admin object, use the following:

```
try
{
   admin.remove_filter (id);
}
catch (FilterNotFound ex) {} // somebody else removed it!
```

To remove all filters from an admin object, use the following:

```
admin.remove_all_filters ();
```

Note that neither of these operations destroys the filter object, they simply remove references to the object.

Event Filters

The filter object itself will not carry out any filtering activities. To create a working event filter, *filter constraints* must be added to the object. A filter can be composed of one or more constraints.

OR semantics are applied between multiple constraints and between multiple filters. If any one constraint in any filter matches the event, the proxy or administration object will forward the event.

Either *AND* or *OR* semantics may be applied between administration object filters and proxy object filters. For OR semantics, an event will be forwarded if it matches either the administration object filters or the proxy object filters. For AND semantics, both must match.

A constraint must be explicitly associated with one or more event types. A constraint will only be evaluated if the event type matches one or more of the event types associated with the constraint. To optimise performance, if no constraints attached to a particular filter match an event's event type the filter will not be invoked at all.

Certain constraints are only applicable to certain types of event. For example, "alarm" events may have "Origin" and "Category" fields in the filterable body while other event types may not. Constraints which filter on Origin and Category fields will only be applicable to "alarm" events.

Constructing Constraints

The following example creates a filter constraint which will pass only events of type *Alarm* from the *Telecom* domain which have a priority greater than *5*.

Step 1: Create an `EventType` array and add the type and domain which will be filtered:

```
EventType types[] = new EventType[1];
types[0] = new EventType ("Telecom", "Alarm");
```

The wildcard character, `*`, can be used in the domain or event type fields if the constraint is to match all event types or domains, as shown in the following code:

```
EventType types1[] = new EventType[1];
types1[0] = new EventType ("*", "*");
```

Step 2: The expression which will filter on priority greater than 5 is a string written using Extended TCL grammar:

```
String expr = "$Priority > 5";
```

Extended TCL is described in *Extended TCL Grammar* on page 56.

Step 3: Create a `ConstraintExp` array to hold the filter constraints created in Steps 1 and 2:

```
ConstraintExp exp[] = new ConstraintExp[1];
exp[0] = new ConstraintExp (types, expr);
```

Step 4: Use the filter object's `add_constraints` operation to attach the constraint to the filter. Each filter object can consist of multiple constraint expressions.

```
try
{
   ConstraintInfo info[] = filter.add_constraints (exp);
   int id = info[0].constraint_id;
```

```
   System.out.println (“Added constraint has id “ + id);
}
catch (InvalidConstraint ex)
{
   System.err.print (“The constraint with the expression “);
   System.err.print (ex.constr.constraint_expr);
   System.err.println (“ is invalid!”);
   // Handle exception.
}
```

Managing Constraints

Each constraint added to a filter is assigned a unique identifier (unique within the scope of that filter object). This provides a means to access specific constraints at run time, allowing them to be modified or deleted.

A filter’s `modify_constraints` operation is used to both modify and delete constraints. The following code demonstrates this. In the example, constraints with identifiers `1`, `2`, `3`, and `5` are deleted and the constraints with identifiers `4` and `6` are modified.

```
int del_list[] = { 1, 2, 3, 5 };
EventType etypes1[] = new EventType[1];
ConstraintExp cexp[] = new ConstraintExp[2];
ConstraintInfo modify_list[] = new ConstraintInfo[2];

etypes1[0] = new EventType (“Telecom”, “Powerfailure”);
cexp[0] = new ConstraintExp (etypes1, “$.voltage < 210”);
modify_list[0] = new ConstraintInfo (cexp[0], 4);

EventType etypes2[] = new EventType[1];
etypes2[0] = new EventType (“Telecom”, “Alarm”);
cexp[1] = new ConstraintExp (etypes2, “$Priority == 3”);
modify_list[1] = new ConstraintInfo (cexp[0], 6);

try
{
   filter.modify_constraints (del_list, modify_list);
}
catch (InvalidConstraint ex)
{
   System.err.print (“The constraint with the expression “);
   System.err.print (ex.constr.constraint_expr);
   System.err.println (“ is invalid!”);
   // Handle exception.
}
catch (ConstraintNotFound ex)
{
   System.err.println (“Constraint with id “ + ex.id + “ not found!”);
   // Handle exception.
}
```

The `modify_constraints` operation can throw an `InvalidConstraint` exception when one of the modified constraints contains invalid syntax. Also, the `ConstraintNotFound` exception is thrown when any of the unique identifiers specified in either of the input sequences cannot be found.

Filters also have a `remove_all_constraints` operation, which removes every constraint added to the filter.

Writing Constraint Expressions

This section describes the syntax and conventions of Extended TCL grammar, which is used for creating filtering constraint expressions.

The following points should be noted if filter performance is an issue:

- Filtering simple data types is faster than filtering complex data types.
- The filter parser uses the `DynAny` interface to process complex data types: this is relatively slow and should be avoided if possible.
- More complex constraint expressions take longer to process.

Extended TCL Grammar

Extended TCL is based on Java-style ‘dot’ notation and syntax. A typical constraint is constructed as follows:

```
$.header.fixed_header.event_type.type_name == 'Info'
```

Keywords are case sensitive in TCL.

The elements used in this expression are individually explained in the following sections.

Basic Elements

$ Token

The `$` token is used to denote the current event. For example, the expression `$domain_name` refers to the value of the current event’s `domain_name` variable, as in the following constraint expression:

```
$domain_name == 'Telecom'
```

The `$` token may refer to either a variable of type `Any` or a variable of type `StructuredEvent`, depending on whether Event Service style or Notification Service style event communication is used.

‘dot’ Operator

The dot operator is used to access an element within a structure. For example, the expression `event_type.type_name` refers to the value of the `type_name` element within the `event_type` structure. The expression `$.remainder_of_body` refers to a field called `remainder_of_body` within the current event.

A full example of a constraint using this operator is:

```
$.header.fixed_header.event_type.type_name == 'Info'
```

Literals

The following literal expressions are allowed within a constraint.

- *Integers:* sequences of digits with optional leading + or -

```
$.header.variable_header(Priority) == 3
```

- *Floats:* sequences of digits with a decimal point and optional exponent notation

```
$.remainder_of_body == 10.5
```

- *Strings*: strings of one or more characters enclosed by single quotation marks: `' '`. To include a single quotation mark in a string, prefix it with a backslash character: `\'`. To include a backslash, use a double backslash: `\\`.

```
$.filterable_data(username) == 'joe'
```

Runtime Variables

Runtime variables are used as shorthand for common components within a structured event. For example, the expression `$.header.fixed_header.event_type.type_name` can be shortened to `$type_name`. Note that there is no dot between the `$` and the variable name in a shortened runtime variable expression.

Runtime variables can be used for any component in the fixed header, variable header, or filterable body of an event. If the runtime variable cannot be found, the expression which uses it defaults to `$.runtime`. This allows generic filters, which can be used for different types of event, to be written.

There is a special runtime variable, `$curtime`, which refers to the current time. Its type is `UtcT` from the `TimeBase` module.

Operators

Comparative Functions

The following comparative operations can be used:

==	equality
!=	inequality
>	greater than
>=	greater than or equal
<	less than

<=	less than or equal
~	substring match
in	element in sequence

The result of applying a comparative function is a boolean value (`true` or `false`).

Example 1

```
$.Cost < 5
```

If the value of the Cost property is less than 5, the expression evaluates to `true`.

Example 2

```
'UK' in $.Country_Name
```

If the `Country_Name` property, which consists of a sequence of strings, includes the string "`UK`", then the expression evaluates to `true`.

Boolean Operators

TCL supports the standard boolean operators `and`, `or`, and `not`. Boolean expressions evaluate to a weakly-typed `long`. This allows complex expressions which evaluate whether a number of boolean expressions are satisfied. For example:

```
$type_name == 'COUNTRY' and (('UK' in $.Country_Name) +
('France' in $.Country_Name) +
('Germany' in $.Country_Name) +
('Italy' in $.Country_Name) +
('Spain' in $.Country_Name)) > 2
```

Special Operators

- The bracket operator, `[ ]`, is used when the component is an array. For example, `$[3]` refers to the fourth element in an event which contains an array.
- A member called `_length` is available when the component is an array or sequence. For example, the expression `$._length > 3` evaluates to `true` for all events that are either arrays or sequences of length four or more.
- The parenthesis operator, `( )`, is used to reference, by name, a particular value within a component that is a list of name-value pairs. For example, `$.header.variable_header (Priority) == 3` evaluates to `true` if the `Priority` QoS in the variable header of a structured event equals 3.
- The `_type_id` member which refers to the unscoped IDL type name. For example, when a component is an IDL struct called `MyEvent`, the `_type_id` field is `MyEvent`.

- The `_repos_id` member which refers to the RepositoryId. For example, when a component is an IDL struct called `MyEvent`, the `_repos_id` field is `IDL:module/MyEvent:1.0`.
- The `default` operator is used when a component is a union, in order to examine whether the union has an active default member or not. For example, the expression `default $` evaluates to `true` when the event is a union with an active default member.
- The exists operator is used to determine whether a field exists within a component or not. For example, `exists $.packets` evaluates to `true` if the event has a field called `packets`.

Mathematical Operators

TCL supports the following mathematical operators:

```
+ - * /
```

Operator Precedence

TCL has the following operator precedence (highest to lowest):

```
( ) exist unary-minus
not
* /
+ - ~
in
== != < <= > >=
and
or
```

Parentheses, `( )`, can be used to over-ride operator precedence.

Constraint Examples

The following examples show constraints that can be used to filter out events based on the values of the event's properties.

These examples assume that structured events of the type created in the example in *Creating Events* on page 39 are being sent.

In each case, the example will pass events for which the constraint expression evaluates to `true`.

- events that have a priority equal to *3*:

```
$.header.variable_header(Priority) == 3
```

- events that have a data value of *42*:

```
$.remainder_of_body == 42
```

- events that have exactly three QoS settings:

```
$.header.variable_header._length == 3
```

- events with data type *long*:

```
$.remainder_of_body._type_id == 'long'
```

- events that time out in less than or equal to three seconds:

```
$.header.variable_header(timeout) <=
$curtime + (3*10*1000*1000)
```

- events which are in the *Telecom* domain and have the *Info* event type:

```
$.header.fixed_header.event_type.domain_name == 'Telecom'
and $.header.fixed_header.event_type.type_name == 'Info'
```

The expression can be simplified using runtime variables (page 57) to give:

```
$domain_name == 'Telecom' and $type_name == 'Info'
```

- all events that do *not* belong to the *Telecom* domain:

```
not $domain_name == 'Telecom'
```

- events that have more than 200 packets or a username called *joe*:

```
$.filterable_data(packets) > 200 or
$.filterable_data(username) == 'joe'
```

2.6 Using Persistence

The Notification Service supports persistent storage via JDBC access to a relational database. Oracle, Sybase, Informix, and hsqldb are supported on both Unix and Windows platforms. Microsoft SQL Server is supported on Windows.

For detailed information on how to configure persistent storage, see the *OpenFusion CORBA Services System Guide*.

3 API Definitions

This section describes selected interfaces and related aspects of the service: the complete IDL API is provided elsewhere as part of the product distribution.

The OMG IDL for version 4 of the OpenFusion Notification Service is the same in as in previous versions, however features which are not supported in version 4 throw a *`NO_IMPLEMENT`* system exception.

3.1 OMG Standard API Definitions

The `CosNotification` module contains common data types and interfaces used throughout the Notification Service. The interfaces in this module are summarized in *Table 1*.

Table 1 CosNotification Interfaces

Interface	Purpose
`AdminPropertiesAdmin`	A base interface for the `EventChannel` interface which supports operations for setting and getting various administrative properties on an event channel object.
`QoSAdmin`	A base interface for the `EventChannel` interface, both administration interfaces, and all of the different proxy interfaces. It supports operations for setting and getting various QoS properties on an event channel and proxy objects. There is also an operation for negotiating the QoS supported by the Notification Service.

The `CosNotifyComm` module contains the client interfaces for the Notification Service. These are the interfaces from which different types of suppliers and consumers need to inherit in order to connect to and communicate with the Notification Service. Note that clients that support interfaces from the `CosEventComm` module can also be connected to the Notification Service. The Notification Service client interfaces are summarized in *Table 2*.

Table 2 CosNotifyComm Interfaces

Interface	Purpose
PushConsumer	An interface for untyped push consumers. The Notification Service version of this interface supports the `PushConsumer` interface from the Event Service as well as the `NotifyPublish` interface.
PushSupplier	An interface for untyped push suppliers. The Notification Service version of this interface supports the `PushSupplier` interface from the Event Service as well as the `NotifySubscribe` interface.
SequencePushConsumer	An interface for sequence style push consumers.
SequencePushSupplier	An interface for sequence style push suppliers. It supports operations for receiving batches of structured events.
StructuredPushConsumer	An interface for structured push consumers.
StructuredPushSupplier	An interface for structured push suppliers. It supports an operation for receiving a structured event.

The `CosNotifyFilter` module contains data types and interfaces used for filtering. The Notification Service supports normal forward filters and so-called mapping filters that can manipulate the priority or timeout values associated with events. The filter interfaces are summarized in *Table 3*.

Table 3 CosNotifyFilter Interfaces

Interface	Purpose
Filter	Interface for a filter. The filter supports match operations for the three different event types as well as operations for managing filter constraints.
FilterAdmin	Interface for filter administrators. This is a base interface for the administration interface and all the proxy interfaces. It supports operations for the management of filter objects.
FilterFactory	Interface for a filter factory. This interface supports operations for creating filter and mapping filter objects.

The `CosNotifyChannelAdmin` module contains the server interfaces for the Notification Service. In particular, there are interfaces for the channel, administration objects and proxy objects. Most of these interfaces extend the corresponding interfaces from the `CosEventChannelAdmin` module in order to make the Notification Service backwards compatible with the Event Service. The interfaces in this module are summarized in *Table 4*.

Table 4 CosNotifyChannelAdmin Interfaces

Interface	Purpose
ConsumerAdmin	An interface for consumer administration objects. The Notification Service version of this interface supports the `ConsumerAdmin` interface from the Event Service as well as the `QoSAdmin`, `NotifySubscribe` and `FilterAdmin` interfaces.
EventChannel	An interface for the event channel. The Notification Service version of this interface supports the `EventChannel` interface from the Event Service as well as the `QoSAdmin` and `AdminPropertiesAdmin` interfaces.
EventChannelFactory	An interface for the event channel factory. The factory supports creation and collection management of event channel objects.

Table 4 CosNotifyChannelAdmin Interfaces (Continued)

Interface	Purpose
ProxyConsumer	A common base interface for proxy consumers. It extends the `QoSAdmin` and `FilterAdmin` interfaces to ensure that all proxy consumers support QoS and filter management.
ProxyPushConsumer	An interface for untyped proxy push consumers. The Notification Service version of this interface is derived from the Event Service `ProxyPushConsumer` and `ProxyConsumer` interfaces.
ProxyPushSupplier	An interface for untyped proxy push suppliers. The Notification Service version of this interface is derived from the Event Service `ProxyPushSupplier` and `ProxySupplier` interfaces.
ProxySupplier	A common base interface for proxy suppliers. It extends the `QoSAdmin` and `FilterAdmin` interfaces to ensure that all proxy suppliers support QoS and filter management.
SequenceProxyPushConsumer	An interface for sequence proxy push consumers. It supports operations for retrieving sequences of structured events.
SequenceProxyPushSupplier	An interface for sequence proxy push suppliers.
StructuredProxyPushConsumer	An interface for structured proxy push consumers. It supports an operation for sending a structured event.
StructuredProxyPushSupplier	An interface for structured proxy push suppliers.
SupplierAdmin	An interface for supplier administration objects. The Notification Service version of this interface supports the `SupplierAdmin` interface from the Event Service as well as the `QoSAdmin`, `NotifyPublish` and `FilterAdmin` interfaces.

Event Channel Factory Interface

The `CosNotifyChannelAdmin::EventChannelFactory` provides functionality for creating new event channels and for getting and listing channels already created by means of the following operations:

- *`create_channel`* - Creates a new event channel with default Quality of Service and administrative settings. The new channel has a unique identifier.
- *`get_all_channels`* - Returns an array of unique identifiers for all channels created by the factory.
- *`get_event_channel`* - Obtains an `EventChannel` object for a given identifier.

Event Channel Interface

The `CosNotifyChannelAdmin::EventChannel` interface extends the corresponding interface from the Event Service as well as the **QoSAdmin** and **AdminPropertiesAdmin** interfaces. In summary, the event channel provides the following operations:

- *`default_consumer_admin`* - This operation returns the default consumer administration object. This object has the unique identification number zero.
- *`default_filter_factory`* - This operation returns the default filter factory.
- *`default_supplier_admin`* - This operation returns the default supplier administration object. This object has the unique identification number zero.
- *`MyFactory`* - This operation returns the factory object that created this event channel object.
- *`for_consumers`* - Event Service style operation for obtaining a `ConsumerAdmin` object. This operation provides backward compatibility with the Event Service and the administration object obtained with this operation does not have a unique identifier.
- *`for_suppliers`* - Event Service style operation for obtaining a `SupplierAdmin` object. This operation provides backward compatibility with the Event Service and the administration object obtained with this operation does not have a unique identifier.
- *`new_for_consumers`* - Preferred way to obtain a `ConsumerAdmin` object with a unique identifier assigned to it.
- *`new_for_suppliers`* - Preferred way to obtain a `SupplierAdmin` object with a unique identifier assigned to it.

- *get_consumeradmin* - Obtains a ConsumerAdmin object for a given identifier. Note that administration objects created with for_consumers cannot be retrieved with this operation.
- *get_supplieradmin* - Obtains a SupplierAdmin object for a given identifier. Note that administration objects created with for_suppliers cannot be retrieved with this operation.
- *get_all_consumeradmins* - Returns a list of unique identifiers for all ConsumerAdmin objects created by this event channel, i.e. by using the new_for_consumers operation.
- *get_all_supplieradmins* - Returns a list of unique identifiers for all SupplierAdmin objects created by this event channel, i.e. by using the new_for_suppliers operation.
- *destroy* - Destroys an event channel.
- *set_qos* - Modifies the quality of service settings of an event channel.
- *get_qos* - Returns the quality of service settings of an event channel.
- *set_admin* - Modifies the administrative settings of an event channel.
- *get_admin* - Returns the administrative settings of an event channel.

The first six of these operations are not described further in this guide as they are either simple *get* operations or else part of the Event Service.

Administration Interfaces

The administration objects, CosNotifyChannelAdmin::ConsumerAdmin and CosNotifyChannelAdmin::SupplierAdmin, are used by both event suppliers and event consumers and serve two distinct purposes:

1. Creating and managing the various proxy objects.
2. Grouping proxies. Both QoS settings and filters set on an administration object are shared by all proxies created by that administration object.

The ConsumerAdmin interface supports additional mapping filter objects that can be used by a client to supersede the priority and timeout QoS settings that an event supplier has defined. This is a useful feature since consumers may have a different view of the relative importance of an event's timeout value from that of the supplier.

The most important functionality of administration objects is to create proxies. Both of the administration interfaces support equivalent operations for creating proxies.

The `ConsumerAdmin` interface operations are listed below. Note that the `SupplierAdmin` interface operations are the same, except that consumer proxies are created instead of supplier proxies:

- *`obtain_push_supplier`* - Event Service style operation for creating a push proxy. Proxies created with this operation are not assigned a unique identifier.
- *`obtain_notification_push_supplier`* - Preferred way to create a push proxy. This operation can create *`Any`* type, *`structured`* type or *`sequence`* type proxies, all of which are assigned a unique identifier.

Filter Interfaces

Filters are objects which can be attached to administration objects and proxy objects. The preferred way to create a filter is by using the filter factory because filters created in this manner are then in the same process as the administration and proxy objects using them. Filter interfaces are defined in the `CosNotifyFilter::Filter`.

The operations for defining filters are located in the `FilterAdmin` interface. These operations are summarised below:

- *`add_filter`* - Attaches a filter to an administration or proxy object. This newly added filter enters the list of filters which are evaluated when the object decides whether or not to forward an event.
- *`remove_filter`* - Removes a filter, with a given identifier, from an administration or proxy object.
- *`get_filter`* - Obtains a filter object for a given identifier.
- *`get_all_filters`* - Returns a list of the unique identifiers for all filters attached to this administration or proxy object.
- *`remove_all_filters`*: Removes all filters attached to this administration or proxy object.

4 Supplemental Information

4.1 Quality of Service Properties

The standard OMG, OpenFusion extended QoS properties, and Administrative Properties are described in detail below.

Standard OMG Properties.

Table 5 lists each of the standard OMG QoS properties, including their associated data types or possible values The four right-hand columns indicate the level (of the channel hierarchy) to which the QoS property may be applied. For example, the `EventReliability` QoS may be applied only at the event channel level or to (structured) events, but not to admin or proxy objects.

Table 5 Standard Quality of Service Properties

Property	Channel	Admin	Proxy	Event
`EventReliability (BestEffort/Persistent)`	×			×
`ConnectionReliability (BestEffort/Persistent)`	×	×	×	
`Priority (short)`	×	×	×	×
`Timeout (TimeT)`	×	×	×	×
`MaxEventsPerConsumer`[1] *`(long)`*	×	×	×	
`OrderPolicy (Any, FIFO, Priority, Deadline)`	×	×	×	
`MaximumBatchSize`[2] *`(long)`*	×	×	×	
`PacingInterval`[2] *`(TimeT)`*	×	×	×	
`DiscardPolicy`[1] *`(Any, FIFO, Priority, Deadline, LIFO)`*	×	×	×	

[1] This QoS property has no meaning when set per supplier admin or per proxy consumer.
[2] At the proxy level, this property only applies to sequence style proxies.

Detailed descriptions of these properties are given below.

EventReliability

The `EventReliability` QoS property controls whether events are delivered using a persistent or a best effort strategy. Setting this property to `Persistent` means that the channel will store events persistently and events are guaranteed to be delivered even when the Notification Service or any of its clients crashes. The default value is `BestEffort`, which means that the Notification Service may lose events during a crash. However, persistent events will be re-delivered to their proxy queues after the crash (proxy queues ignore events that have already been delivered to the connected consumer).

The persistence of events is managed by the event database plugin. The Notification Service supports different plugin modules to support different application requirements. Please consult the *System Guide* for details on configuring the persistent plugin.

ConnectionReliability

The `ConnectionReliability` QoS property controls whether connections are handled using a persistent or a best effort strategy.

Note that setting event reliability to persistent and connection reliability to best effort is a combination that has no meaning and is not supported. The default value is `BestEffort`, which means that connections will be lost when the Notification Service fails to deliver or receive events from a client.

All clients should also be implemented as persistent objects when the `ConnectionReliability` QoS property is to be set to `Persistent`. The reason for this is that client objects need to assume the same identity when recovered after a crash. This is the only way that the Notification Service can logically reconnect to the client. The Notification Service will never be able to reconnect to a transient client object.

The Notification Service will keep retrying persistent client objects until an `OBJECT_NOT_EXIST` system exception is encountered. This exception is raised by an object activator when the client object no longer exists. The `MaxReconnectAttempts` QoS property, described later, may be used to limit the durability of persistent clients.

Priority

The `Priority` QoS property defines the relative priority of an event: the higher the number, the higher the priority. It is normally set in the variable header of a structured event. The priority may also be set on a per-channel, per-admin or per-proxy basis. Applying the priority to an event channel object means that all events that pass through the channel will receive that priority unless another value is

set in the variable header. The default priority of an event is `zero`. The event priority QoS applies only when the `OrderPolicy` and `DiscardPolicy` QoS properties have a value of `PriorityOrder`.

Timeout

The `Timeout` QoS property defines a relative timeout for an event. It is normally set in the variable header of a structured event. The Notification Service deletes this event from all queues when this timeout occurs. A consumer views an expired event in the same way as it does an event that was never delivered to the Notification Service.

The unit for the `Timeout` QoS is 100 nanoseconds and the default value is `zero`, which means that no timeout is applied. A value in the range of 1-9999 is not supported, i.e. the smallest value for the event timeout is one millisecond. The lowest value is used when both the `Timeout` and the `StopTime` QoS are defined for an event.

The event timeout QoS is always applicable. It can be used further when the `OrderPolicy` and `DiscardPolicy` QoS properties have a value of `DeadlineOrder`.

The timeout may also be set on a per-channel, per-admin or per-proxy basis. Applying the timeout to an event channel object means that all events that pass through the channel will receive the said timeout value unless a value is set in the variable header.

MaxEventsPerConsumer

The `MaxEventsPerConsumer` QoS property defines the maximum number of events that a proxy will queue on behalf of the connected consumer. This setting can be used to prevent a single consumer from exhausting the master queue. The default queue size for `MaxEventsPerConsumer` is unlimited (its `property` value is set to zero).

The `MaxEventsPerConsumer` QoS property applies to the proxy queues. QoS properties may be fine grained or coarse grained so each proxy queue may have different maximum queue length, or all proxies that are created by one consumer administration object may have the same maximum queue lengths.

The `MaxEventsPerConsumer` QoS property is typically used when the incoming event rate exceeds the capabilities of the Notification Service for extended periods of time. It is also used when the proxy queue represents periodic updates that will be available in the shape of a new event at a later time. Limiting the queue size also reduces the resources required by the Notification Service.

OrderPolicy

The `OrderPolicy` QoS property defines the order in which events are delivered. The default value is `PriorityOrder`, which means that events are delivered according to their priority. The Notification Service applies a `FifoOrder` policy for delivering events with the same priority. The other settings for this QoS are `DeadlineOrder` and `AnyOrder`. The `DeadlineOrder` policy means that events with the shortest timeout value will be delivered first.

`OrderPolicy` has no meaning when applied to supplier admins or proxy consumers. Attempting to set this QoS on a supplier admin or proxy consumer will have no effect (but will produce a warning in the service log).

MaximumBatchSize

The `MaximumBatchSize` QoS property controls the maximum number of events a sequenced event consumer will receive for each event delivery. The default value is one, i.e. a sequence type consumer will receive one event at a time. A sequence consumer would normally always increase this value since having a batch size of one defeats the performance advantage of using sequencing.

PacingInterval

The `PacingInterval` QoS property defines the maximum time a sequence type client will wait between subsequent event deliveries. A value set to zero means that the consumer is willing to wait until such time as `MaximumBatchSize` events are available. The unit for this QoS is 100 nanoseconds and the default value is zero. A value in the range 1-9999 is not supported, i.e. the smallest value for the pacing interval is one millisecond. Note that the consumer will always wait until at least one event is available.

DiscardPolicy

The `DiscardPolicy` QoS property defines the order in which events are discarded from event queues. The following values determine the order that events are discarded.

- *`AnyOrder`* - any event may be discarded when the queue becomes full.
- *`FifoOrder`* - the first event received will be the first discarded.
- *`PriorityOrder`* - events will be discarded in priority order such that the lower priority events will be discarded before the higher priority events. The order in which events of the same priority are discarded is determined by the `PriorityDiscardPolicy` setting.
- *`DeadlineOrder`* - events will be discarded in the order of the shortest expiry deadline will be discarded first.

The default value for `DiscardPolicy` is `AnyOrder`.

The discard policy is not used by the master queue when the `RejectNewEvents` administrative property is set to `TRUE`.

Events are discarded from the master queue when the value of the `MaxQueueLength` administrative property is reached. An event that is discarded from the master queue will never reach any consumer and appears to the consumer as though the event was never delivered to the event channel.

Events are discarded from proxy queues once the value of the `MaxEventsPerConsumer` QoS is reached. The other settings for this QoS are `PriorityOrder`, `DeadlineOrder`, `FifoOrder`, and `LifoOrder`.

The Notification Service is able to optimise queues when they:

- use the same order and discard policies
- when the order policy is the same and the discard policy is set to `AnyOrder`

The service must maintain separate orderings when different order and discard policies are used.

OpenFusion QoS Extensions

Table 6 lists the QoS properties provided in the OpenFusion Notification Service to extend the OMG Notification Service standard QoS properties.

Table 6 Extended Quality of Service Properties

Property	Channel	Admin	Proxy	Event
MaxReconnectAttempts[1]*(long)*	×	×	×	
ReconnectInterval[2] *(TimeT)*	×	×	×	
ConnectedClient[2] *(Object)*			×	
MaxInactivityInterval[3]*(TimeT)*	×	×	×	
AutoSequenceBatchSize (long)	×	×	×	
AutoSequenceTimeout (ulonglong)	×	×	×	
DisconnectCallback	×	×	×	
MaxMemoryUsage	×			
MaxMemoryUsagePolicy	×			
PropagateQoS	×	×		

[1]This QoS property applies only to proxy push suppliers.

[2]This QoS property is read only.
[3]This QoS property applies only to proxy pull suppliers and proxy push consumers.

Detailed descriptions of these properties are given below.

MaxReconnectAttempts

The `MaxReconnectAttempts` QoS property defines the maximum number of times the Notification Service will attempt to reconnect to a failed pull supplier or push consumer. The Notification Service disconnects the client as though the disconnect operation had been invoked on the proxy when the client is still unavailable after the maximum number of attempts have been made.

Theoretically, the *absolute timeout value* for pull suppliers and push consumers is the product of the `MaxReconnectAttempts` property value and the `ReconnectInterval` property value. However, the *actual* time taken for the entire timeout period can take longer than the absolute timeout value:

1. The `ReconnectInterval` property is the interval of time the Notification Service will wait before making another connection attempt. This interval is measured from the time that it *becomes aware* that a connection attempt failed (e.g. by receiving an exception from the ORB).

2. The absolute timeout value cannot account for the length of time taken from when a client disconnection occurs until the time that the Notification Service becomes aware of the disconnection. Normally, this is not an issue, but under certain circumstances (such as when the orb daemon is not running on particular ORBs) the effect of this delay can be dramatic.

   For example, if an ORB takes 20 seconds to pass an exception indicating client disconnection, then the `ReconnectInterval` will effectively be increased by 20 seconds. Assuming that the `ReconnectInterval` is set to 1 second and the number `MaxReconnectAttempts` is set to 120, then the actual absolute timeout will be 120 * (20+1) = 2520 seconds = 42 minutes, instead of the expected 120 seconds (2 minutes).

ReconnectInterval

The `ReconnectInterval` QoS property defines the interval of time that the Notification Service will wait before retrying persistent pull suppliers and push consumers that are unavailable. This interval is measured from the time that it *determines* that a connection attempt failed (see *MaxReconnectAttempts* above).

This QoS property has no meaning when `ConnectionReliability` is set to `BestEffort`. Also note that this QoS has no meaning for push suppliers and pull consumers.

The Notification Service waits for the reconnect interval before resuming event reception or delivery after event communication has failed. The unit for this QoS is 100 nanoseconds and the default value is one second, i.e. 10,000,000 nanoseconds. A value in the range 1-9999 is not supported, i.e. the smallest value for the reconnect interval is one millisecond.

The Notification Service considers an event consumer or supplier to be unavailable when the operation that retrieves or delivers events raises a system exception. The only system exception is the `OBJECT_NOT_EXIST` exception and this is handled differently to other system exceptions by the Notification Service, i.e. the proxy object is disconnected when a client raises this exception.

ConnectedClient

The `ConnectedClient` QoS property is a read-only property that applies only to proxy objects. The value associated with this QoS is the object reference of the client associated with the proxy. For example, the `ConnectedClient` QoS property contains a structured push consumer object for structured push supplier proxies.

MaxInactivityInterval

The `MaxInactivityInterval` QoS property is the connection timeout for push suppliers. This is a relative timeout value and is reset whenever a supplier calls `push` on its consumer regardless of whether the operator is successful or not; in other words, the timeout is reset when the proxy detects any activity from its client.

When the proxy has been idle for the maximum inactivity interval, then the Notification Service disconnects the client as though the disconnect operation had been invoked on the proxy.

The unit for `MaxInactivityInterval` is 100 nanoseconds. The default value is 0 (zero), which disables this QoS and allows idle push suppliers and pull consumers to never timeout. The minimum supported timeout value (other than the zero default value) is one millisecond, i.e. values of 10000 or greater.

AutoSequenceBatchSize

The maximum batch size that will be sent by a structured proxy (consumer or supplier) when auto-sequencing is being used. When the proxy has received this number of events, they will be sent as a single batch. The default value is 200 events. If the `AutoSequenceTimeout` interval is exceeded while the proxy is waiting for sufficient events to complete a batch, the batch will be sent even if it is incomplete.

To disable auto-sequencing, set this QoS to 0 or 1, *or* set `AutoSequenceTimeout` to a value less than 10.

See *Auto-sequencing* on page 24 for more information about auto-sequencing.

AutoSequenceTimeout

This is the maximum amount of time that will be allowed to elapse before an auto-sequence batch is sent. If this interval elapses before the batch reaches the required size (specified by the `AutoSequenceBatchSize` property), the incomplete batch is sent regardless.

The unit for this property is milliseconds. The default value is 200 milliseconds.

To disable auto-sequencing, set this QoS to a value less than 10 *or* set `AutoSequenceBatchSize` to 0 or 1.

See *Auto-sequencing* on page 24 for more information about auto-sequencing.

DisconnectCallback

This property affects all proxies. If set to true (the default) then when a proxy's disconnect method is called, then the disconnect method on its connected client will also be called. This behaviour is in accordance with the behaviour specified in the OMG Notification Service Specification v1.3.

If set to false, then a proxy's connected client will not have its disconnect operation invoked when that of the proxy is invoked. This behaviour is in accordance with the behaviour specified in the OMG Notification Service Specification v1.0.

MaxMemoryUsage

Affects the memory size of event channels. `MaxMemoryUsage` is similar in purpose to the property `MaxQueueLength`, except that the size of memory is controlled, rather than the number of events. `MaxMemoryUsage` takes a value of type `ulonglong`. The units for this property are *bytes*. When this value is exceeded then attempts will be made to limit memory usage according to the current usage policy. The current usage policy is controlled using the `MaxMemoryUsagePolicy` property.

MaxMemoryUsagePolicy

Affects event channels. `MaxMemoryUsagePolicy` is the policy by which memory usage is controlled when MaxMemoryUsage is exceeded. It can take one of three values:

- *`PurgeEvents`* - If this value is set, then `MaxMemoryUsage` is treated as a soft limit. Whenever an event is received that pushes memory usage above the `MaxMemoryUsage` level, that event will be added to the internal queue of the appropriate event channel as normal. Then, in a manner that mirrors discard behaviour, the event at the back of the queue will have its data purged from

memory. If the event is set to *best effort* delivery, then it is effectively discarded and the memory it used will be available for recovery by the garbage collector. However, in the case of a persistent event a place holder will remain in memory so that the data can be reloaded from its persistent store, when required. Therefore, in the case of a persistent event, not all of the memory used will freed and the total memory usage will continue to increase. Nonetheless, the rate of increase will be greatly reduced making this an appropriate policy for dealing with bursts of event delivery.

Note that if events contain very small amounts of data then very little memory will be recovered by purging them, as it is the event data that is purged from memory. *PurgeEvents* will produce better results with larger event sizes.

- *DiscardEvents* - If this value is set, then MaxMemoryUsage is treated as a limit. Whenever an event is received that takes memory usage above MaxMemoryUsage, an event is discarded according to the current discard policy. Note that since events vary in size, the memory usage may still grow since the new event may be larger than that which is discarded.
- *RejectEvents* - If this value is set, then MaxMemoryUsage is treated as a hard limit. Whenever an event is received that takes memory usage above MaxMemoryUsage, an org.omg.CORBA.IMP_LIMIT exception is thrown.

The default value of this property is *PurgeEvents*.

PropagateQoS

Controls how changes to a QoS on an event channel are propagated to admins and proxies.

When PropagateQoS is set to *false* (the default), changes made to a QoS after it has been set on a channel will not affect the QoS settings on an admin or proxy. When it is set to *true*, changes made to the QoS on the channel will carry through to the admins and proxies, even over-riding any QoS that has been set individually on the admin or proxy.

For example, the Timeout QoS is set to *10000* on the event channel. This setting is applied to all admins and proxies created on that channel. If Timeout is then changed to *20000* on the channel while PropagateQoS is set to false, the admins and proxies retain their setting of *10000*. Any new admins and proxies, however, will take on the new value of *20000*.

If Timeout is changed to *20000* on the channel while PropagateQoS is set to *true*, the admins and proxies also take on the new setting of *20000*.

Administrative Properties

Administrative properties refer to property settings that may be applied *only* to event channel objects. These properties are usually set when an event channel is first created. These settings are typically static in nature although they may be changed during the lifetime of the channel object. The standard administrative properties are described below.

MaxQueueLength

The `MaxQueueLength` administrative property defines the maximum size of the *master queue* for an event channel. The value of the `MaxQueueLength` property should normally be greater than any value of a `MaxEventsPerConsumer` QoS property.

This prevents any badly-behaved consumer (for example a consumer that consumes events very slowly or a consumer that remains suspended for an extended period of time) from causing events to be rejected from the master queue. The maximum possible size of the master queue is the accumulative size of all proxy queues.

Normally, the size of the master queue is smaller than the accumulative size of all proxy queues because there is typically an overlap in the events received by different consumers.

MaxConsumers

The `MaxConsumers` administrative property defines the maximum number of consumers that can be concurrently connected to an event channel. The consumers are counted as all the proxy suppliers of all the consumer administration objects managed by the event channel.

MaxSuppliers

The `MaxSuppliers` administrative property defines the maximum number of suppliers that can be connected concurrently to an event channel. The suppliers are counted as all the proxy consumers of all the supplier administration objects managed by the event channel.

RejectNewEvents

The `RejectNewEvents` administrative property indicates whether events should be rejected or discarded, according to the `DiscardPolicy` setting, when the `MaxQueueLength` for the master queue is exceeded. The `RejectNewEvents` property can have the following values:

- *TRUE*: New events received by the event channel are rejected when the `MaxQueueLength` is exceeded. A push supplier encounters an `IMP_LIMIT` system exception when it attempts to deliver an event to the channel. Also, the event channel will not issue any pull operations on pull suppliers until the queue size has been reduced.
- *FALSE*: New events received by the event channel are discarded according to the `DiscardPolicy` QoS setting when the maximum queue length is exhausted. Both push and pull suppliers can keep delivering events to the channel, but this may cause some events to be discarded.

The `RejectNewEvents` administrative property, when set to *true*, guarantees that the Notification Service will never drop any events.

4.2 Errors and Exceptions

Errors

The Notification Service improves on the Event Service by providing QoS settings that define how to deal with most runtime errors. Events are stored persistently when the `EventReliability` QoS setting is set to persistent and the service fails. All persistent events are recovered and re-delivered to all registered clients once the Notification Service is restarted after the service has crashed.

Also, the Notification Service keeps trying its connections when the `ConnectionReliability` QoS setting is set to persistent until it encounters an `OBJECT_NOT_EXISTS` exception. The Notification Service just starts delivering all queued events when a client crashes but is later restored with the same object reference as it had when first connecting to the Notification Service.

How events are removed from the internal queues of the Notification Service is defined by the `DiscardPolicy` QoS setting. Events are discarded when either the `MaxQueueLength` or `MaxEventsPerConsumer` values are exceeded. Note that the service keeps storing un-delivered events until the system resources are exhausted when there is no limit on the queue length.

Exceptions

The Notification Service supports a number of exceptions which are summarised in *Table 7*.

Table 7 Notification Service Exceptions

Exception	Description
`AdminLimitExceeded`	Indicates that the limit for the number of concurrently connected proxies has been exceeded.
`AdminNotFound`	Indicates that the administration object with the specified unique identifier was not found in an event channel.
`AlreadyConnected`	Indicates that a consumer or supplier was already connected.
`CallbackNotFound`	Indicates that a callback object with the specified unique identifier was not found in a filter.
`ChannelNotFound`	Indicates that the channel with the specified unique identifier was not found in an event channel factory.
`ConnectionAlreadyActive`	Indicates that a connection was already active and an attempt was made to resume it.
`ConnectionAlreadyInactive`	Indicates that a connection was already inactive when an attempt was made to suspend it.
`ConstraintNotFound`	Indicates that a constraint with the specified unique identifier was not found in a filter.
`Disconnected`	Indicates that a disconnected client is trying to send or receive the event.
`DuplicateConstraintID`	Indicates that a sequence of constraints contain duplicate unique constraint identifiers.
`FilterNotFound`	Indicates that the filter object with the specified unique identifier was not found in an administration or proxy object.
`InvalidConstraint`	Indicates that a constraint set on a filter object was invalid.

Table 7 Notification Service Exceptions (Continued)

Exception	Description
InvalidEventType	Indicates that an event type is not supported or is invalid.This exception is not thrown by the OpenFusion Notification Service.
InvalidGrammar	The grammar specified was not `EXTENDED_TCL`, SQL92, or the name of a valid Filter class name.
InvalidValue	Indicates that a constraint value is invalid, e.g. when a priority value is not of type `short` or when a timeout value is not of type `TimeT`.
ProxyNotFound	Indicates that the proxy object with the specified unique identifier was not found in an administration object.
TypeError	Indicates a type error.
UnsupportedAdmin	Indicates that an administrative setting on an event channel was not supported.
UnsupportedFilterableData	Indicates that an event contains data which could not be processed by a filter object. This exception is normally not propagated back to clients.
UnsupportedQoS	Indicates that a quality of service setting on an event channel, administration or proxy object was not supported.

Implementation Limit Exception

The CORBA specification provides a general exception, *org.omg.CORBA.IMP_LIMIT*, for indicating when a limit has been reached or exceeded. This exception is raised by the Notification Service, specifically, when an event is pushed to a proxy push consumer and either:

1. the value of the QoS property *MaxQueueLength* has been reached and the QoS property *RejectNewEvents* is set to true
2. any resource, such as threads or memory, which is insufficient, exhausted, or unavailable

The `org.omg.CORBA.IMP_LIMIT` exception includes important information in its exception message. For example, in the case of sequence proxy push consumers, the exception message contains the number of events that were accepted by the

Notification Service (from the sequence) before the exception was raised. This information is important, since it can be used to ensure that the same events are not unnecessarily supplied more than once to the Notification Service. In addition to the number of events accepted, the message also contains other information, such as the limit exceeded and the length of the supplied sequence.

i

The `org.omg.CORBA.IMP_LIMIT` exception stores the number of accepted events in the last three hexadecimal digits of its minor code *provided* that the length of the supplied sequence is less than or equal to 0xFFF (4096): the number may be extracted from the minor code by subtracting the base PrismTech minor code of 0x50540000 from its value.

This feature can be used to avoid the overhead of string manipulation which is otherwise needed to obtain the information from the exception message.

Event Type Repository

5 Description

5.1 Overview

The OpenFusion implementation of the Notification Services includes the Event Type Repository, which is an optional feature specified by the OMG.

The Event Type Repository contains meta-data about event types. The repository contains information about the properties of an event for each event type. The repository contains information only about the properties in the filterable body of a structured event because it was specifically designed to fulfil the requirement of verifying filter constraints.

An important property of the Event Type Repository is the ability to modify the event types and the relationship between event types at runtime. This allows applications to evolve over time, e.g. an application can create a new event type, with additional properties, that inherits from an existing event type. New applications can take advantage of the additional information, while existing applications process the event according to the old set of properties.

5.2 Concepts and Architecture

Figure 10 shows the UML model for the Event Type Repository. The repository is a singleton that supports a number of event domains and contains a number of event types. An event type in turn has a domain, a name and a number of properties. Event types can inherit or import other event types.

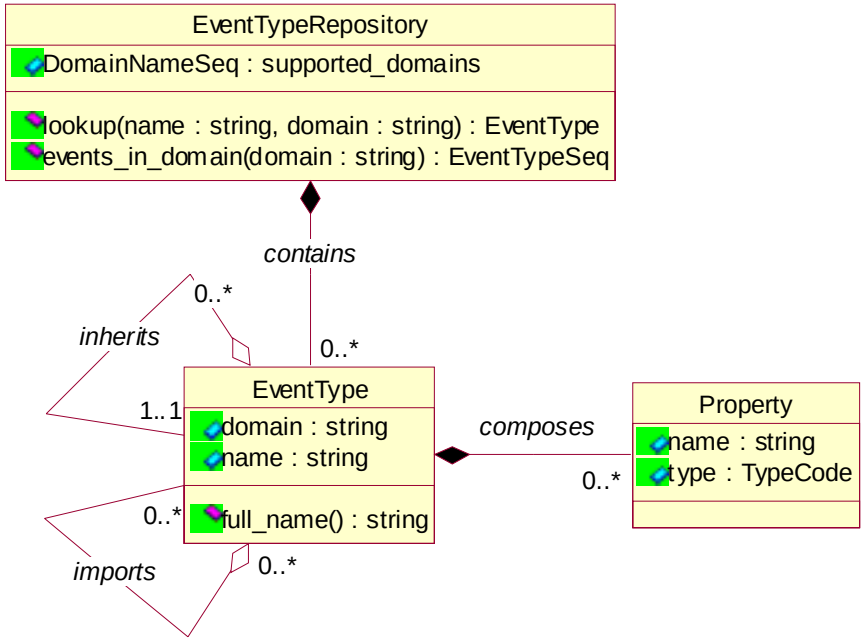

Figure 10 Event Type Repository Model

The Event Type Repository model shown in *Figure 10* is mapped to IDL using the guidelines set out in the Meta Object Facility (MOF). The most important thing to realise about the mapping is that links are transformed into interfaces rather than operations. In addition to this, each class has a meta class with some standard operations. Finally, the mapping automatically adds a package class and a meta class for the package class.

Event Types

An event type is defined by three components (refer to *Figure 10*):

- a *domain name* - a high level categorisation of the event, for example *Telecom* and *Transport* are domain names
- a *type name* - categorises events within a domain

- a *sequence of properties* - a sequence of name-value pairs where *name* states the property's name, and *value* states the type of property it is when associated with an event type

Inheritance

An event type can inherit the properties of another event type. This means that all the properties in the super type will also be present in the sub type. Also, the creation of inheritance cycles is not allowed.

Importing

One event type can be imported into another in addition to inheritance. This does not create an inheritance relationship but all the properties of the imported type will be present in the importer type. Property names may overlap but only when the type associated with the property is the same in both the imported and importer event types.

Contains

The Event Type Repository is populated with event types using the `Contains` interface. It is possible for clients to look up event types and investigate what properties are available for filtering once populated. Thus, clients can use the repository to create meaningful constraint expressions for event filtering.

Interfaces

The Event Type Repository consists of twelve interfaces. Section 7, *API Definitions*, on page 115. The operations from these interfaces provide a generic way to reflect on an object or association.

6 Using Specific Features

This section describes how to use the following specific features of the Event Type Repository:

- *Adding an Event Type*: describes how to add a new event to the repository.
- *Properties*: demonstrates the management of event type properties.
- *Event Types*: demonstrates the management of event types.
- *Composition*: demonstrates how to associate and disassociate properties with/from event types.
- *Inheritance*: demonstrates how to create inheritance relationships between event types.
- *Import*: demonstrates how to create import relationships between event types.
- *Event Type Repository description*: explains how to use the event type repository.
- *Containment*: demonstrates how to add and remove event types from the event type repository.
- *Repository package*: explains how to use the package interfaces.

Note:

- All interfaces in the Event Type Repository inherit other interfaces from the Meta Object Facility (MOF). This section is not intended to a reference to MOF operations. Accordingly, the MOF `RefObject` and `RefAssociation` interface operations are not described here.
- Most operations in the Event Type Repository do not accept arguments containing a null value. The OpenFusion implementation checks the input arguments and raises a `BAD_PARAM` exception when a null argument is encountered. The examples used below assume that input values are not null, and therefore, this exception is not checked.

Import Statements

The following packages must be imported into any application which use the Event Type Repository:

```
org.omg.Reflective.*
org.omg.NotificationTypes.*
```

The `Reflective` and `NotificationTypes` packages include exception definitions from the MOF plus all of the interfaces and data types from the Event Type Repository.

6.1 Adding an Event Type

This topic briefly introduces some of the Event Type Repository interfaces. Common tasks when using the repository are to create an event type, add some properties to it and then add the event type to the repository. The first task in using the repository is to resolve and create it. This task is shown in the listing below:

```
org.omg.CORBA.Object object = null;
RefBaseObject ref = null;
_NotificationTypesPackage pack = null;
EventTypeRepository repos = null;

try
{
   object = orb.resolve_initial_references (“NotificationTypes”);
}
catch (org.omg.CORBA.ORBPackage.InvalidName ex)
{
   System.err.println (“Failed to resolve Event Type Repository”);
   ex.printStackTrace ();
   System.exit (1);
}

repos = EventTypeRepositoryHelper.narrow (object);
ref = repos.repository_container ();
pack = _NotificationTypesPackageHelper.narrow (ref);
```

The Event Type Repository in this code is resolved and used to obtain a specific package object. The package object has a reference to the following objects:

- An object that implements the `EventTypeRepositoryClass` interface. This can be used to create a new event type repository.
- An object that implements the `EventTypeClass` interface. This object can be used to create new `EventType` objects.
- An object that implements the `PropertyClass` interface. This object can be used to create new `Property` objects.
- Four objects that can be used to manipulate the different aggregations in the event type repository model. The objects implement the `Contains`, `Inherits`, `Imports`, and `Composes` interfaces, respectively.

The variables `pack` and `repos` are class variables that are used in the following to obtain the link interfaces and manipulate the repository. The listing below shows how to create a property and an event type. Once these have been created, the property is added to the event type and the event type is finally added to the event type repository.

```
// Get relevant object references.
PropertyClass property   = pack.property_class_ref ();
EventTypeClass eventType = pack.event_type_class_ref ();
Composes composes        = pack.composes_ref ();
Contains contains        = pack.contains_ref ();
```

```
Property p1 = null;
EventType type = null;

org.omg.CORBA.TypeCode tc;

// Create a property.

try
{
   tc = orb.get_primitive_tc (org.omg.CORBA.TCKind.tk_string);
   p1 = property.create_property (“Operator”, tc);
}
catch (SemanticError ex)
{
   System.err.println (“Failed to create property!”);
   ex.printStackTrace ();
   System.exit (1);
}

// Create a new event type.

try
{
   type = eventType.create_event_type (“telecom”, “alarm”);
}
catch (SemanticError ex)
{
   System.err.println (“Failed to create event type!”);
   ex.printStackTrace ();
   System.exit (1);
}

// Add property to event type.

try
{
   composes.add (type, p1);
}
catch (StructuralError ex)
{
   System.err.println (“Can’t add property to event type!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}

// Add event type to repository.

try
{
   contains.add (repos, type);
}
catch (StructuralError ex)
{
   System.err.println (“Can’t add event type to repository!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
```

```
}
```

The above code is relatively straightforward. Relevant objects are first obtained from the package object. The code then performs the following steps:

Step 1: Creates a property using an object that implements the `PropertyClass` interface.

Step 2: Creates an event type in the **`telecom`** domain with type **`alarm`** using an object that implements the `EventTypeClass` interface.

Step 3: Adds the property to the event type using the `Composes` interface.

Step 4: Adds the event type to the repository using the `Contains` interface.

6.2 Properties

A property is an object that encapsulates a name and a type code. The name of a property is linked to the name of a property in the filterable body of a structured event and the type code determines the value type (but not the actual value) of the property.

Properties are created using the factory meta class `PropertyClass`. See *Adding an Event Type* on page 90 for obtaining a reference to an object that implements the `PropertyClass` interface by means of the package object.

The `PropertyClass` interface has two additional operations besides the factory operation for creating property objects. These are a result of the MOF mapping from the meta model to IDL:

- *`all_of_kind_property`*: This operation returns all properties, excluding any subtypes of the `Property` class. This operation returns just the properties that have been created by the factory as the repository meta model does not have any subtypes of the `Property` class.
- *`all_of_type_property`*: This operation returns all properties, including any subtypes of the Property class. This operation returns all the properties that have been created by the factory in a similar manner to the `all_of_kind_property` operation.

The example below shows how to use the `create_property` operation to create a new property:

```
PropertyClass factory = pack.property_class_ref ();
Property p1 = null;
org.omg.CORBA.TypeCode type;

try
{
   type = orb.get_primitive_tc (org.omg.CORBA.TCKind.tk_string);
   p1 = factory.create_property (“User”, type);
}
```

```
catch (SemanticError ex)
{
   System.err.println (“Failed to create property!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The name and type code of a property can be obtained and set once created. Note that any property can be used as a factory for creating other properties since the `Property` interface inherits from the `PropertyClass` interface. The example below shows how to print the name and type code of a property:

```
try
{
   String name = p1.name ();
   org.omg.CORBA.TypeCode tc = p1.type_code ();
   System.out.println (“name=” + name + “, type=” + tc);
}
catch (Exception ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

Note that the get operations on the `Property` interface are allowed to raise both the `StructuralError` and `SemanticError` exceptions. The OpenFusion implementation of the Event Type Repository does not raise any exceptions on the `get` operations.

It is also possible to set a new name for a property and to change the type code. However, a few restrictions apply:

- The new property name must not be used by an existing property for the event type or any super type of the event type when the property has already been added to an event type.
- The new property name must only be present in the *import graph* of the event type when the type code is the same when the property is added to an event type. Import graph means the event type itself or any event type imported by the event type. Note that the super types of that event type are also part of the import graph when an event type is imported.

The example below shows how to set the name and type code of a property:

```
try
{
   p1.set_name (“Data”);
}
catch (SemanticError ex)
{
   System.err.println (“Name already used!”);
   ex.printStackTrace ();
   System.exit (1);
}

try
```

```
{
   org.omg.CORBA.TypeCode tc;
   tc = orb.get_primitive_tc (org.omg.CORBA.TCKind.tk_long);
   p1.set_type_code (tc);
}
catch (SemanticError ex)
{
   System.err.println (“Illegal type code!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The `SemanticError` exception is raised when the new name or the new type code conflicts with another property in the event type inheritance and import hierarchy. This exception is not raised when the property has yet to be added to an event type.

6.3 Event Types

An event type is an object that can be added to the event type repository. It describes the expected contents of the filterable body field of a structured event. This description is divided into three components:

- The properties of the event type itself.
- The properties in the super types of the event type. This includes all the event types imported by any super type.
- The properties in any type imported by the event type. This includes all the event types in any super types of an imported event type.

The inheritance and import hierarchies defined above are referred to as the *complete graph*. The complete graph for an event type defines all the properties that are expected in the filterable body of a structured event of that event type.

Event types are created using the `EventTypeClass`. See *Adding an Event Type* on page 90 for obtaining a reference to an object that implements the `EventTypeClass` interface. The event type factory contains two operations to list all objects created as does the `PropertyClass` interface. These are not described any further here.

An `EventType` object can be created as follows:

```
EventTypeClass factory = pack.event_type_class_ref ();
EventType type = null;

try
{
   type = factory.create_event_type (“telecom”, “ring”);
}
catch (SemanticError ex)
{
   System.err.println (“Illegal type name!”);
   ex.printStackTrace ();
   System.exit (1);
```

```
}
```

Note that the factory operation raises a `SemanticError` exception when the event type name has a length of *zero*. The `EventType` interface inherits the `EventTypeClass` interface in a similar manner to the `Property` interface. This means that all event type objects can be used as factories as well.

The `EventType` interface has five operations. There are three *get* operations to obtain the domain name, the type name, and the full name. The full name of an event type is composed of the names of all super types and the usual name separated by dots. In addition, there are two *set* operations to set the domain and type. An example of using the *get* operations is shown below:

```
try
{
   System.out.println (“Domain    = “ + type.domain ());
   System.out.println (“Name      = “ + type.name ());
   System.out.println (“Full name = “ + type.get_full_name ());
}
catch (Exception ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The *get* operations are all allowed to raise `StructuralError` and `SemanticError` exceptions according to the interface, but these exceptions are never raised by the OpenFusion implementation.

As for the `Property` interface, there are a few restrictions related to using the *set* operations:

- The `SemanticError` exception is raised by the `set_domain` operation when the repository does not support the new domain and the event type has been added to an event type repository.
- The `set_name` operation raises a `SemanticError` exception when an event type with that name already exists and the event type has already been added to a repository.

The example below shows how to use the *set* operations of the `EventType` interface:

```
try
{
   type.set_domain (“transport”);
}
catch (SemanticError ex)
{
   System.err.println (“Domain not allowed in repository!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

```
try
{
   type.set_name (“alarm”);
}
catch (SemanticError ex)
{
   System.err.println (“Event type already exists!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

6.4 Composition

Creating properties or event types in isolation is not very useful. This section describes how to create associations between event types and properties using the `Composes` interface. An object that implements the `Composes` interface can be obtained by means of the package interface.

The `Composes` interface has a number of operations for adding, removing and modifying the properties associated with an event type. The interface also has operations for obtaining information about which properties and event types are associated with the event type. The query operations are summarised below:

- *`all_Composes_links`*: This operation returns all the links that are currently established between properties and event types. Two elements in the sequence are returned by this operation: a property and an event type.
- *`exists`*: This operation simply checks that an association between an event type and a property exists.
- *`with_composition`*: This operation returns all the properties that have been associated with a particular event type.
- *`with_component`*: This operation returns the event type that is associated with a particular property.

Note that these operations are present in all the *link* interfaces (with slightly different names) due to the MOF mapping. The following example code listings illustrate how to use these operations. Code examples are provided for only the `Composes` interface since these operations are similar for all the link interfaces.

```
Composes composes = pack.composes_ref ();
ComposesLink[] cl = composes.all_Composes_links ();
EventType type = null;
Property prop = null;

try
{
   for (int i = 0; i < cl.length; i++)
   {
      type = cl[i].composition;
      prop = cl[i].component;

      System.out.println (“Link #” + i + “:”);
```

```
      System.out.println (“Event domain :” + type.domain ());
      System.out.println (“Event name   :” + type.name ());
      System.out.println (“Property name:” + prop.name ());
      System.out.println (“Property type:” + prop.type_code ());
   }
}
catch (Exception ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The `all_Composes_links` operation is more likely to be used by a browser tool than by an application, but it may be useful, for example, for getting a full listing of all the associations of a `Composes` object. Using the `exists` operation is very straightforward:

```
if (composes.exists (type, prop))
{
   System.out.println (“Property is added to event type”);
}
else
{
   System.out.println (“Property isn’t added to event type”);
}
```

In the above example, it is assumed that the variables `type` and `prop` are event type and property objects created elsewhere in the code. Next, the `with_composition` operation is called to get all the properties associated with an event type:

```
Property[] props = composes.with_composition (type);

try
{
   System.out.println (“Event domain :” + type.domain ());
   System.out.println (“Event name   :” + type.name ());
   System.out.println (“Properties:”);
   for (int i = 0; i < props.length; i++)
   {
      System.out.println (“Property name:” + props[i].name ());
      System.out.println (“Property type:” + props[i].type_code ());
   }
}
catch (Exception ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

Again, the `type` variable is assumed to be an event type defined elsewhere in the code. Finally, the `with_component` operation can be used to find the event type that has a particular property associated with it:

```
EventType et = composes.with_component (prop);

try
{
   System.out.println (“Property name:” + prop.name ());
   System.out.println (“Property type:” + prop.type_code ());
```

```
   System.out.println (“Is associated with the event type:”);
   System.out.println (“Event domain :” + type.domain ());
   System.out.println (“Event name   :” + type.name ());
}
catch (Exception ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The remaining operations of the `Composes` interface deal with associations between properties and event types and are summarised below:

- *`add`*: Adds a property to an event type.
- *`add_before_component`*: Adds a property to an event type at a particular position.
- *`modify_composition`*: Moves a property from one event type to another event type.
- *`modify_component`*: Replaces one property in an event type with another property.
- *`remove`*: Removes a property from an event type.

The following shows an example of using the `add` operation:

```
try
{
   composes.add (type, prop);
}
catch (StructuralError ex)
{
   System.err.println (“Failed to add property to event type!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The `add` operation adds the property at the end of the list of properties of an event type since properties are ordered. The `StructuralError` exception is raised in the following circumstances:

- when the property is already added to this event type,
- when any super type has a property with this name added, and
- when the property has a different type code in `any` event type in the import graph.

The `add_before_component` operation is used when you wish to place a new property at a particular position in the ordered list of properties. An example is given below:

```
try
{
   composes.add_before_component (type, prop, before);
}
catch (NotFound ex)
{
   System.err.println (“Could’t find before property!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println (“Failed to add property to event type!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

This code is very similar to the plain `add` example. Note that the `NotFound` exception is raised when the `before` property is not associated with `type`. As with the previous examples, it is assumed that the `type` variable is an event type, whereas `prop` and `before` are properties that have been created or obtained previously in the program.

The `modify_composition` operation is used to move a property from one event type to another event type. This operation essentially deletes the property from one event type and adds it to another. Here is an example of how the operation is used:

```
try
{
   composes.modify_composition (type, prop, new_type);
}
catch (NotFound ex)
{
   System.err.println (“Property was not added to event type!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println (“Failed to add property to new event type!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The `NotFound` exception is raised when the property is not associated with the event type in a similar manner to the `add_before_component` operation. The `StructuralError` exception is raised in the following circumstances:

- when the property is already added to the new event type,
- when any super type of the new event type has the property added, or
- when the property has a different type in any event type in the import graph of the new event type.

A property can also be replaced with another using the `modify_component` operation rather than by moving a property from one event type to another. This operation removes one property from an event type and adds another as shown below:

```
try
{
   composes.modify_component (type, prop, new_prop);
}
catch (NotFound ex)
{
   System.err.println (“Property was not added to event type!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println (“Failed to add property to event type!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

Again, `NotFound` is raised when the property is not associated with the event type. The usual causes for the `StructuralError` exception apply:

- the new property is already added to the event type
- any super type of the event type has this property added
- the property has a different type in any event type in the import graph of the event type

Finally, the `remove` operation can be used to delete, i.e. disassociate, a property from an event type. The use of this operation is fairly straightforward as with most operations in the `Composes` interface:

```
try
{
   composes.remove (type, prop);
}
catch (NotFound ex)
```

```
{
   System.err.println ("Property was not added to event type!");
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println ("Never raised!");
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println ("Never raised!");
   ex.printStackTrace ();
   System.exit (1);
}
```

Note that this implementation of the `Composes` interface never checks that an event type has subtypes or that any other event types import the event type. This means that the property is also deleted from any subtype of the event type, and from any importer type when a property is deleted from an event type.

Adding a property to an event type automatically adds the property to any subtype and any importer type of the event type in a similar manner to the above. The operations in the `Composes` interface should therefore be used with caution when modifying the properties in an existing event type hierarchy.

6.5 Inheritance

This section describes how to use the operations in the `Inherits` interface to create or modify inheritance hierarchies of event types.

The `Inherits` interface contains the same query operations as the `Composes` interface because it represents an aggregation (or a *link*) in the repository meta model:

- *`all_Inherits_links`* - This operation returns all the inheritance relationships that are currently established between event types. The elements in the sequence that are returned by this operation contain two elements: a subtype and a supertype.
- *`exists`* - This operation simply checks whether one event type inherits from another.
- *`with_sub_type`* - This operation returns the event type with a particular subtype.
- *`with_super_type`* - This operation returns all the event types that inherit from a particular supertype.

These query type operations are not described any further here because code examples have previously been provided.

The remainder of this section provides examples using the rest of the operations in the `Inherits` interface. Note that the operations themselves are rather similar to those in the `Composes` interface. The reason for this is that they are both *links* mapped to IDL using the MOF.

This section therefore emphasises the circumstances that cause an exception to be raised, rather than the operations themselves. Below is a summary of the operations for manipulating inheritance hierarchies between event types:

- *`add`* - Creates an inheritance relationship between two event types.
- *`modify_sub_type`* - Replaces one subtype with another.
- *`modify_super_type`* - Replaces one supertype with another.
- *`remove`* - Deletes an inheritance relationship between two event types.

There is no operation for adding one object before another since the inheritance between event types is not ordered. Note that only single inheritance between event types is allowed in the Event Type Repository.

The `add` operation creates an inheritance relationship between two event types. An example is shown below:

```
EventTypeClass factory = pack.event_type_class_ref ();
EventType sub_type = null;
EventType super_type = null;

try
{
   sub_type = factory.create_event_type (“telecom”, “alarm”);
   super_type = factory.create_event_type (“telecom”, “location”);
}
catch (SemanticError ex)
{
   System.err.println (“Illegal type name!”);
   ex.printStackTrace ();
   System.exit (1);
}

Inherits inherits = pack.inherits_ref ();

try
{
   inherits.add (sub_type, super_type);
}
catch (StructuralError ex)
{
   System.err.println (“Couldn’t add subtype to supertype!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

In the above, two event types are created in the usual fashion. An object that implements the `Inherits` interface is resolved using the package as described earlier. An inheritance relationship is created between the two event types using this interface. The `StructuralError` exception is raised in these circumstances:

- When the subtype is already added to another supertype. This is due to the fact that the Event Type Repository supports only single inheritance between event types when:
- any property in the subtype is defined in the supertype event type
- any property type in the subtype is defined in any type in the import graph of supertype
- adding this event type creates a cycle in the inheritance hierarchy

The `modify_sub_type` operation is used to replace one subtype with another. It is a shorthand for first deleting one event type from a supertype and then adding another event type to the supertype. An example is shown below:

```
try
{
   inherits.modify_sub_type (sub_type, super_type, new_sub_type);
}
catch (NotFound ex)
{
   System.err.println (“Subtype wasn’t added to supertype!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println (“Couldn’t replace subtype!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The `StructuralError` exception is raised in the same circumstances as noted above because the `modify_sub_type` operation creates an inheritance relationship between two event types, i.e. adds an event type to another. The `NotFound` exception is raised when the subtype has not been added to the supertype.

The supertype can also be modified using the `modify_super_type` operation. Again, this is a shorthand method for first removing an inheritance relationship between two event types and then creating another. This is illustrated below:

```
try
{
   inherits.modify_super_type (sub_type, super_type, new_super_type);
}
catch (NotFound ex)
```

```
{
   System.err.println ("Subtype wasn't added to supertype!");
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println ("Couldn't replace supertype!");
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println ("Never raised!");
   ex.printStackTrace ();
   System.exit (1);
}
```

The `NotFound` exception is raised when an inheritance relationship does not exist between the subtype and supertype. The `StructuralError` exception is raised in the following circumstances:

- when a subtype has any properties that are defined in the new super type or any of its supertypes
- when a subtype has any properties that are defined in the import graph of the new supertype
- when adding this event type creates a cycle in the inheritance hierarchy

Finally, the `remove` operation is used to delete an inheritance relationship between two event types. Its use is straightforward:

```
try
{
   inherits.remove (sub_type, super_type);
}
catch (NotFound ex)
{
   System.err.println ("Subtype wasn't added to supertype!");
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println ("Never raised!");
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println ("Never raised!");
   ex.printStackTrace ();
   System.exit (1);
}
```

The `NotFound` exception is again raised when the subtype does not inherit from the supertype. Both the `StructuralError` and `SemanticError` exceptions are not raised by this implementation of the `Inherits` interface.

The same note of caution stated for composition applies to inheritance. The implementation of the `Inherits` interface does not check for existing relationships when the inheritance hierarchy is modified. As an example, an entire branch of the tree may be moved by invoking the `modify_sub_type` operation.

6.6 Import

Event types can import rather than inherit properties from other event types. An import relationship between two event types just means that one event type obtains the properties of another event type. There is no semantic relationship. This section shows how to use the operations in the `Imports` interface.

The query operations of the `Imports` interface are summarised as follows:

- *`all_Imports_links`* - This operation returns all the import relationships that are currently established between event types. The elements in the sequence that are returned by this operation contain two elements: importer type and imported type.
- *`exists`* - This operation simply checks whether one event type imports another.
- *`with_importer`* - This operation returns all the event types that import a particular event type.
- *`with_imported`* - This operation returns all the event types that are imported by a particular event type.

The remaining operations of the `Imports` interface are summarised below:

- *`add`* - Creates an import relationship between two event types.
- *`modify_importer`* - Moves an imported event type from one importer event type to another.
- *`modify_imported`* - Replaces an imported event type with another.
- *`remove`* - Deletes an import relationship between two event types.

The `add` operation is used to create an import relationship. The two sides of the relationship are called the importer event type and the imported event type respectively. An imported event type may have overlapping property names as long as the type codes of the properties are the same, unlike with inheritance.

```
EventTypeClass factory = pack.event_type_class_ref ();
EventType importer = null;
EventType imported = null;

try
{
   importer = factory.create_event_type (“telecom”, “alarm”);
   imported = factory.create_event_type (“telecom”, “location”);
```

```
}
catch (SemanticError ex)
{
   System.err.println (“Illegal type name!”);
   ex.printStackTrace ();
   System.exit (1);
}

Imports imports = pack.imports_ref ();

try
{
   imports.add (importer, imported);
}
catch (StructuralError ex)
{
   System.err.println (“Failed to import event type!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The `Imports` object reference is resolved from the package object in the usual manner. The `StructuralError` exception is raised in the following circumstances:

- any property in the event type graph of the imported event type has a different type code than the corresponding property in the importer event type
- the addition of the event type creates a cycle in the import graph.

An example of using the `modify_importer` is shown below. This operation is a shorthand method for first removing the imported type from one event type and subsequently adding it to another event type.

```
try
{
   imports.modify_importer (importer, imported, new_importer);
}
catch (NotFound ex)
{
   System.err.println (“Event type wasn’t imported!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println (“Failed to import event type!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The `NotFound` operation is raised when an import relationship between the two event types does not exist as has been the case for many of the previous link manipulating operations. The `StructuralError` exception is raised in the same circumstance as stated above because the imported event type is added to a new importer.

The `modify_imported` operation replaces an imported event type with another. The operation is a shorthand method for first removing an event type from an importer and then adding a new event type to the same importer. An example is shown below:

```
try
{
   imports.modify_imported (importer, imported, new_imported);
}
catch (NotFound ex)
{
   System.err.println (“Event type wasn’t imported!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println (“Failed to import event type!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The `NotFound` and `StructuralError` exceptions are raised in the same circumstances as for the `modify_importer` operation.

Finally, the `remove` operation destroys an import relationship between two event types. Again, the `NotFound` exception is raised when no import relationship exists between the two types:

```
try
{
   imports.remove (importer, imported);
}
catch (NotFound ex)
{
   System.err.println (“Event type wasn’t imported!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
```

```
   ex.printStackTrace ();
   System.exit (1);
}
```

6.7 Event Type Repository Description

An Event Type Repository is an object that contains a number of event type objects. The repository supports a number of domains that constrain the domain names of the events that are added to the repository.

An Event Type Repository is a singleton object within each server. The repository object is typically created by the OpenFusion server process but can alternatively be created using the factory meta class `EventTypeRepositoryClass`. The `EventTypeRepositoryClass` interface has two additional operations besides the factory operation for creating property objects:

- *`all_of_kind_event_type_repository`* - This operation returns either the event type repository singleton or a sequence of length *`zero`* since only one repository is allowed within each package.
- *`all_of_type_event_type_repository`* - As above, this operation returns one or zero event type repositories.

The example below shows how to use the `create_event_type_repository` operation to create a new repository:

```
EventTypeRepositoryClass etc;
EventTypeRepository repos = null;
String domains[] = { “oil”, “banking”, “”, “finance” };

etc = pack.event_type_repository_class_ref ();

try
{
   repos = etc.create_event_type_repository (domains);
}
catch (AlreadyCreated ex)
{
   System.err.println (“Repository already created!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Failed to create repository!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The `AlreadyCreated` exception is raised when an attempt is made to create multiple repositories within the same server. The `EventTypeRepository` interface has two additional operations to those inherited from the `EventTypeRepositoryClass` interface:

- *supported_domains* - This returns a list of strings describing the domains that are supported by the repository.
- *lookup* - This operation locates an event type with a particular type name and domain.

As the supported_domains operation is very simple, this section includes example code for only the lookup operation:

```
EventType type = null;

try
{
   type = repos.lookup (“alarm”, “telecom”);
}
catch (InvalidName ex)
{
   System.err.println (“Invalid type name!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (TypeNotFound ex)
{
   System.err.println (“Event type not found!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (UnknownDomain ex)
{
   System.err.println (“Domain not supported by repository!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}

try
{
   System.out.println (“Full name = “ + type.get_full_name());
}
catch (Exception ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

6.8 Containment

The last step in configuring the Event Type Repository is the Contains interface. It allows event types to be added to and removed from the repository. As with the other link interfaces, there are four query operations:

- *all_Contains_links* - This operation returns all the containment relationships that are currently established between the event type repository and the event types. The elements in the sequence returned by this operation contain two elements: the repository and an event type.
- *exists* - This operation simply checks that an event type has been added to the repository.
- *with_container* - This operation returns all the event types in the repository.
- *with_contained* - This operation returns the repository where an event type is defined. This will always be the singleton event type repository of the package.

The Contains interface also has the following additional operations for manipulating relationships between event types and the repository:

- *add*: Adds an event type to the repository.
- *modify_container* - Moves an event type from one repository to another.
- *modify_contained* - Replaces an event type in the repository with another.
- *remove* - Removes an event type from the repository.

An example of using add is shown below:

```
Contains contains = pack.contains_ref ();

try
{
   contains.add (repos, type);
}
catch (StructuralError ex)
{
   System.err.println (“Can’t add event type to repository!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

In the above example, it is assumed that type is an event type created or obtained previously in the program. The StructuralError exception is raised when the event type has already been added to the repository or when the domain of the event is not supported. The SemanticError is not raised by this implementation of the event type repository.

It is possible to have multiple repositories by creating multiple packages although the event type repository is a singleton within each package. The `modify_container` operation is not needed when your application uses only a single repository. However, below is an example of how to move an event type from one repository to another:

```
try
{
   contains.modify_container (repos, type, new_repos);
}
catch (NotFound ex)
{
   System.err.println (“Event type wasn’t in repository!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println (“Failed to add event type to repository!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

The `NotFound` exception is raised when the event type is not added to the repository. The `StructuralError` exception is raised when the event type could not be added to the new repository, i.e. when the domain is not supported or when it is already added.

The `modify_contained` operation replaces one event type in the repository with another. It is a shorthand method for first deleting one event type and subsequently adding another. An example of usage is listed below:

```
try
{
   contains.modify_contained (repos, type, new_type);
}
catch (NotFound ex)
{
   System.err.println (“Event type wasn’t in repository!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println (“Failed to add new event type to repository!”);
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println (“Never raised!”);
   ex.printStackTrace ();
   System.exit (1);
}
```

Finally, the `remove` operation deletes an event type from the repository. The code below is straightforward:

```
try
{
   contains.remove (repos, type);
}
catch (NotFound ex)
{
   System.err.println ("Event type wasn't in repository!");
   ex.printStackTrace ();
   System.exit (1);
}
catch (StructuralError ex)
{
   System.err.println ("Never raised!");
   ex.printStackTrace ();
   System.exit (1);
}
catch (SemanticError ex)
{
   System.err.println ("Never raised!");
   ex.printStackTrace ();
   System.exit (1);
}
```

The `remove` operation raises only the `NotFound` exception when no containment relationship exists between the event type and the event type repository.

6.9 Repository Package

The `Package` interfaces section provides only a brief overview of the operations for similar reasons as before.

There is a `NotificationTypesPackageFactory` interface for creating package instances. The package factory is used to create a local instance of an event type repository. Note that there can still be only a single repository per server even when used in this way:

```
_NotificationTypesPackage pack;
EventTypeRepositoryClass etc;
EventTypeRepository repos;
NotificationTypesPackageFactoryImpl impl;
NotificationTypesPackageFactory factory;
org.omg.CORBA.Object object;

String[] domains = { "Telecom", "Transport", "News" };

impl = new NotificationTypesPackageFactoryImpl ();
object = ObjectAdapter.getObject (impl);
factory = NotificationTypesPackageFactoryHelper.narrow (object);

try
{
   pack = factory.create_notification_types_package ();
   etc = pack.event_type_repository_class_ref ();
   repos = etc.create_event_type_repository (domains);
}
catch (org.omg.Reflective.SemanticError ex)
{
```

```
   System.err.println ("Semantic error occurred!");
   ex.printStackTrace ();
   System.exit (1);
}
catch (org.omg.Reflective.AlreadyCreated ex)
{
   System.err.println ("Local repository already created!");
   ex.printStackTrace ();
   System.exit (1);
}
```

The factory interface has just a single operation for creating packages: `create_notification_types_package`. A package is an object that implements the `NotificationTypesPackage` interface. This interface has operations to obtain references to all the objects described previously:

- *`property_class_ref`* - Returns a `PropertyClass` factory object for this package.
- *`event_type_class_ref`* - Returns an `EventTypeClass` factory object for this package.
- *`event_type_repository_class_ref`* - Returns a factory object of type `EventTypeRepositoryClass` for this package.
- *`contains_ref`* - Returns the `Contains` object.
- *`inherits_ref`* - Returns the `Inherits` objects.
- *`imports_ref`* - Returns the `Imports` object.
- *`composes_re`* -: Returns the `Composes` object.

Previous topic have already describe how to use these simple get operations.

7 API Definitions

The Event Type Repository consists of the 12 interfaces, described in this section. The operations from these interfaces provide a generic way to reflect on an object or association.

The Event Type Repository interfaces provide specific operations in order to access the functionality of the repository so the generic, reflective operations are not needed. As an example, the `Composes` interface has an `add` operation that adds a property to an event type. There is also an `add_link` operation that is inherited from the `RefAssociation` interface. These operations perform the same action, one in a domain-specific way and one in a generic way.

Classes in the UML model inherit operations from the `RefObject` interface. *Table 8* shows the Event Type Repository interfaces that deal with classes.

Table 8 Event Type Repository Classes

Interface	Purpose
`NotificationTypesPackageClass`	A package level interface that can be used to create event type repository packages.
`NotificationTypesPackage`	Instances of the event type repository package that are created by the factory class.
`EventTypeRepositoryClass`	A meta class and factory for objects that implement the `EventTypeRepository` interface.
`EventTypeRepository`	An interface for event type repositories.
`EventTypeClass`	A meta class and factory for objects that implement the `EventType` interface.
`EventType`	An interface for the event type objects contained in the repository.
`PropertyClass`	A meta class and factory for objects that implement the `Property` interface.
`Property`	An interface for property objects. Event type objects are composed of property objects.

Links (aggregations in the UML model) inherit operations from the `RefAssociation` interface. *Table 9* shows the Event Type Repository interfaces that deal with links.

Table 9 Event Type Repository Aggregations

Interface	**Purpose**
Contains	An interface for manipulating the contents of an event type repository. The interface represents the aggregation between the repository and the event type classes in *Figure 10*.
Inherits	An interface for manipulating inheritance between event types. The interface represents the self-aggregation on the event type class in *Figure 10*.
Imports	An interface for manipulating imports between event types. The interface represents the self-aggregation on the event type class in *Figure 10*.
Composes	An interface for manipulating compositions between event types and properties. The interface represents the aggregation between the event type and property classes in *Figure 10*.

8 Supplemental Information

8.1 Exceptions

The Event Type Repository supports a number of exceptions as summarised in *Table 10* below:

Table 10 Event Type Repository Exceptions

Exception	Description
InvalidName	Indicates that an event name was invalid.
UnknownDomain	Indicates that the event type repository does not know a domain.
TypeNotFound	This exception is only raised by the `lookup` operation of the `EventTypeRepository` class to indicate that an event type could not be found.

A number of exceptions from the MOF are used in addition to the Event Type Repository exceptions. These are summarised in *Table 11* below:

Table 11 MOF Exceptions used by the Event Type Repository

Exception	Description
StructuralError	Raised when an operation could not complete because it would result in a structural error, e.g. the repository would be inconsistent.
SemanticError	Indicates a semantic error. This is raised when a check of the input parameters shows that the operation cannot be performed.
NotFound	Indicates that an object could not be found in a container.
AlreadyCreated	Indicates that an Event Type Repository has already been created. This exception is raised because the repository is a singleton.

Event Domain Service

9 Description

9.1 Overview

The Event Domain Service simplifies the federation and management of Notification Service event channels.

The *Management of Event Domains* specification was developed by the OMG Telecom Task Force. It describes standard interfaces for federating and managing a set of Notification Service Event Channel objects, or a set of Log Service Log objects. The OpenFusion implementation of the Event Domain Service is wholly compliant with the OMG specification.

The federation of event channels using the Notification Service can be cumbersome and involve several steps. The same operation using the Event Domain Service can be performed in a single step.

The Event Domain Service can manage the following types of objects:

- Notification-style event channels
- Notification-style typed event channels
- Log Service logs
- Log Service typed logs

Although the Event Domain Service can manage notification channels and logs, it is independent of these other services. It is a stand-alone service that can be used to manage objects from any OMG-compliant Notification or Log Service.

Service Features

The Event Domain Service provides the following features:

- Networking of event channels:
  - facilitates channel federation
  - no need for an intermediary client to forward events from one channel to another
  - also supports typed events and log domains
- Simplified programming:
  - federate channels in one operation
  - connect a client in one operation

- Ability to detect and (if necessary) prevent the creation of *cycles* and *diamonds* — which helps in topology management.

9.2 Architecture and Concepts

An *event domain* is a group of one or more channels. (The term *channel* is used here to denote any managed object, including Notification Service event channels and Log Service logs.) The channels within a domain may or may not be connected to one another (*federated*).

Note that the channels need not belong to the same Notification Service instance. Channels from different Notification Services can be federated into a single domain. Similarly, logs from different Log Service instances can be federated into a single log domain.

Event suppliers and consumers can connect to any channel in the event domain, using the operations provided by the Event Domain Service interfaces. Events flow through the domain, from supplier to consumer, and may pass through any number of federated channels in the process.

Each event domain may optionally have default supplier and consumer channels specified. An event supplier will connect to the default supplier channel unless a specific target channel is identified when the connection is established. An event consumer will connect to the default consumer channel unless a specific target channel is identified when the connection is established.

Figure 11 illustrates events flowing through one possible configuration of an event domain with four event channels.

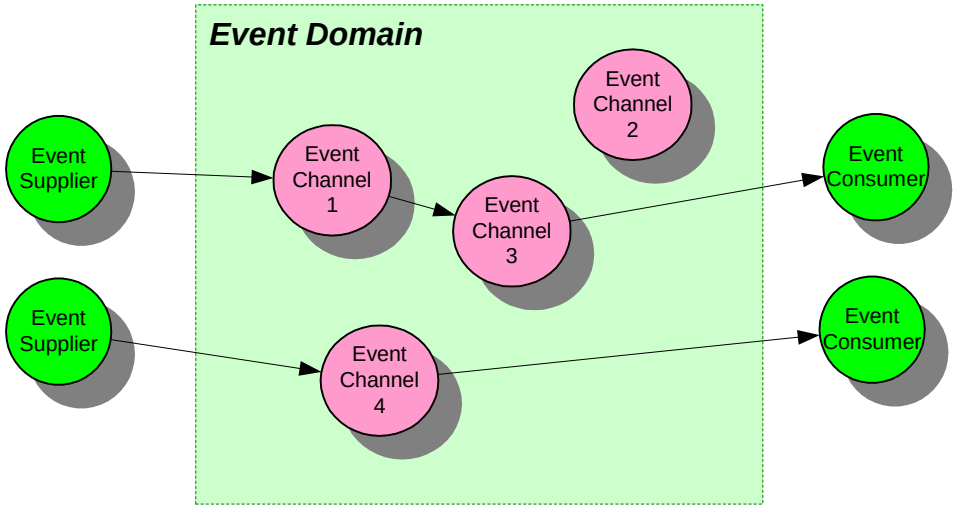

Figure 11 An Event Domain

The Event Domain Service does not interfere with the events that flow inside a domain. It is merely a management service that facilitates the administrative tasks associated with federating and managing channels.

It is possible to have any mixture of connection types and event propagation models within a single domain. For example, the connection between one set of channels may be `structured` using the pull model, while the connection between another set of channels may be `untyped` using the push model.

Federating Channels

A powerful feature of the Event Domain Service is the ability to federate channels without the use of an intermediary. In other words, it is possible to connect two channels without creating a special client that forwards events from one channel to the other.

An example of federated Notification Services is shown in *Figure 12*. The ability to federate event channels in this manner provides improved flexibility (alternative paths can be made available), scalability (the system can be easily extended), and better performance.

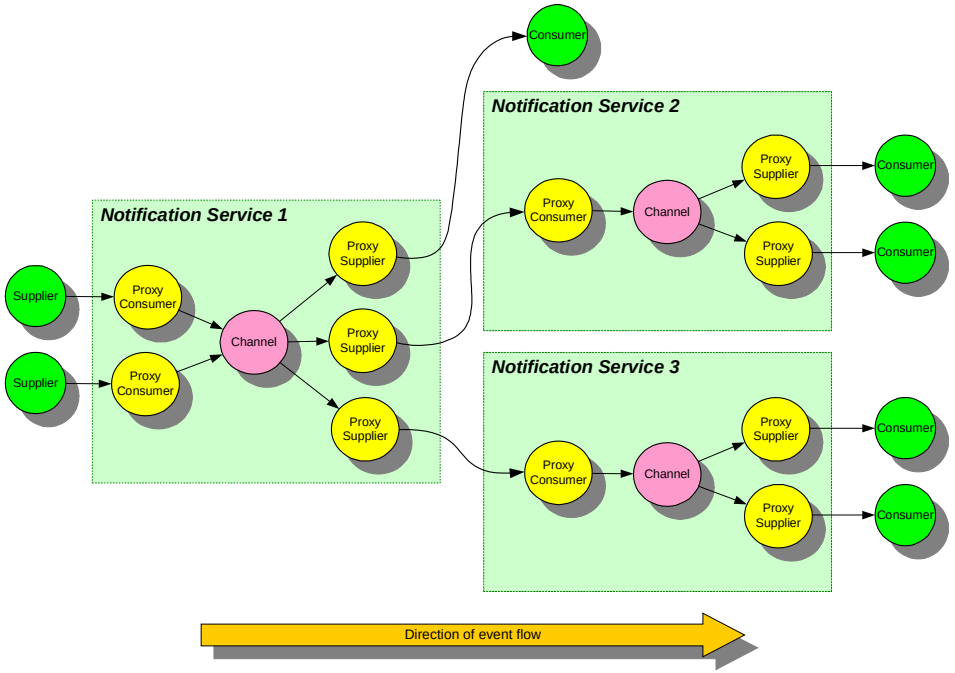

Figure 12 Federated Notification Service Example

The Event Domains Service provides interfaces and operations that allow the federation of an event channel in a single operation. Consider the connection between *Notification Service 1* and *Notification Service 2* in *Figure 12*. When using the interfaces of the Notification Service, the following steps are needed to establish this connection:

1. Get a reference to the source event channel in *Notification Service 1*.
2. Get or create a consumer admin object for this channel.
3. Obtain a proxy supplier from the consumer admin object.
4. Get a reference to the target event channel in *Notification Service 2*.
5. Get or create a supplier admin object for this channel.
6. Obtain a proxy consumer from the supplier admin object.
7. Connect the proxy supplier by passing in the proxy consumer object.
8. Connect the proxy consumer by passing in the proxy supplier object.

Using the interfaces of the Event Domain Service, this procedure can be replaced by a single operation.

Event Type Propagation

The OpenFusion implementation of the Event Domain Service supports a QoS setting for enabling or disabling *event type propagation* in a domain. An event type change can cause a large number of callbacks in a network of many channels, so some applications may want to disable event type propagation for performance reasons.

Domain Topology

The *topology* of an event domain describes the way in which event channels are connected together within the domain. These connections can be illustrated as a *channel graph* (see *Figure 11* on page 123 and *Figure 13* on page 126 for examples of channel graphs).

The nature of the Notification Service implies that a connection is always directed. Thus, for any channel in the domain, it is possible to define *upstream* and *downstream* directions of event flow. For example, in *Figure 11* on page 123 event channel 1 is upsteam from event channel 3, while event channel 3 is downstream from event channel 1.

The Event Domain Service does not enforce any restrictions on how channels should be connected. Channels may be connected to multiple other channels in the domain. Some channels may be part of an event domain and yet not be connected to any other channel in the domain.

Figure 13 shows four different types of event domain that can be created:

1. a domain where the channels are connected as a directed acyclic graph
2. a domain that contains a diamond
3. a domain that contains a cycle
4. a domain where only some of the channels are connected

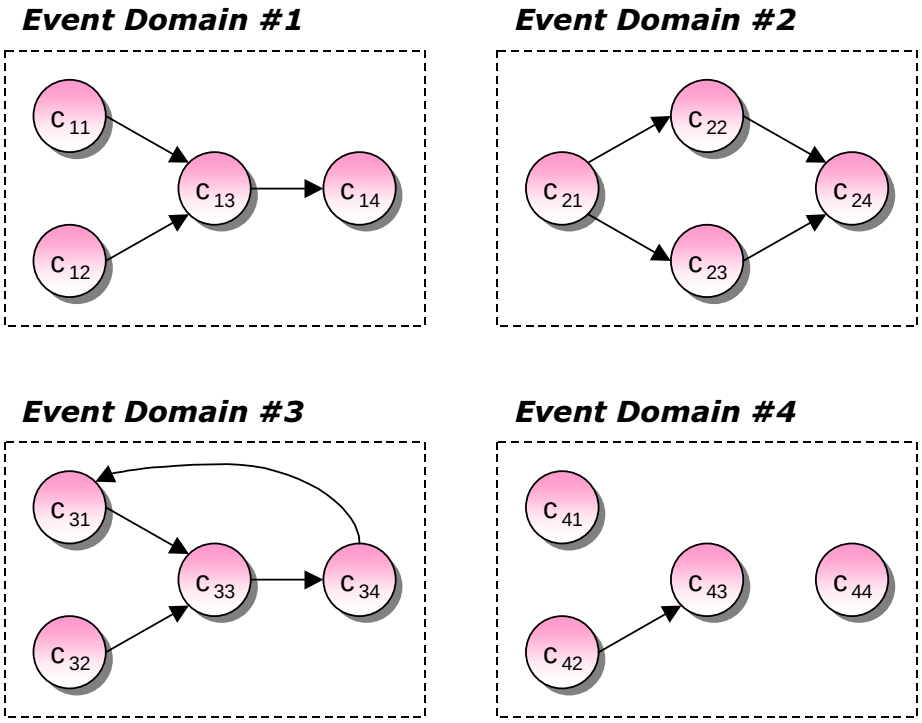

Figure 13 Different Types of Event Domains

Elaborate domain topologies can be constructed which contain combinations of these features. For example, a domain could contain both a cycle and a diamond, or diamonds with multiple paths. Such complex topologies might be required to provide redundancy in the case of a channel failure, for example. But the presence of diamonds and cycles can cause effects which the developer must be aware of and possibly take steps to avoid when the event domain is designed:

- A cycle may cause an event to propagate endlessly within the domain. To avoid this, appropriate filters can be set up in the Notification Service or an event timeout can be set.
- A diamond may cause an event to be delivered more than once to an end consumer (the number of times being equal to the number of alternative paths by which the event may arrive at the consumer). To avoid redundant event deliveries, appropriate filters can be set up in the Notification Service.

The Event Domain Service supports two standard Quality of Service (QoS) properties which can be used to prevent the creation of diamonds or cycles:

- *Cycle detection* rejects any attempt to create a connection between two channels when the resulting channel graph would contain a cycle.
- *Diamond detection* rejects any attempt to create a connection between two channels when the resulting channel graph would contain a diamond.

Gathering Topology Information

The Event Domain Service supports two operations for obtaining information about cycles and diamonds that may exist in a domain:

- *get_cycle*`s` returns a sequence of cycles in a specified domain.
- *get_diamond*`s` returns a sequence of diamonds in a specified domain.

Two additional operations can be used to obtain information about the topology of an event domain:

- *get_offer_channels* returns an array of channel identifiers for all channels upstream from a specified channel.
- *get_subscription_channels* returns an array of channel identifiers for all channels downstream from a specified channel.

10 Using Specific Features

This section uses simple examples that work through the interfaces and describe how to use the individual operations of the Event Domain. Related operations are grouped together for clarity. Additional examples for using the service, complete with source code and descriptions of how to compile and run them, are supplied elsewhere as part of the product distribution.

Section *10.1* is a simple end-to-end example which sets up an event domain and connects it to an event supplier and an event consumer. Sections *10.2* and *10.3* expand on this and describe domain management operations for untyped and typed event domains, respectively. Section *10.4* describes how these features can be applied to log domains.

Notes

- There is little or no error-checking in the examples shown here. Code to deal with exceptions has generally been omitted for the sake of clarity and brevity. These exceptions must be properly caught and handled in a working system.
- These examples use features of the OpenFusion Naming Service to register and resolve object names. This is purely for convenience: it is not necessary to use the Naming Service with the Event Domain Service.

Import Statements

The examples provided below use the following packages:

```
org.omg.CosNotifyComm.*
org.omg.CosNotification.*
org.omg.CosNotifyChannelAdmin.*
org.omg.CosEventDomainAdmin.*
org.omg.CosTypedEventDomainAdmin.*
```

10.1 Setting up a Domain

The following examples will set up the event domain shown in *Figure 14*. This simple domain consists of three Notification Service event channels, labelled *A*, *B*, and *C* in the diagram.

The setup client is responsible for creating the domain, creating the channels and adding them to the domain, and creating the connections between the channels. References to all created objects will be placed in the root context of the OpenFusion Naming Service.

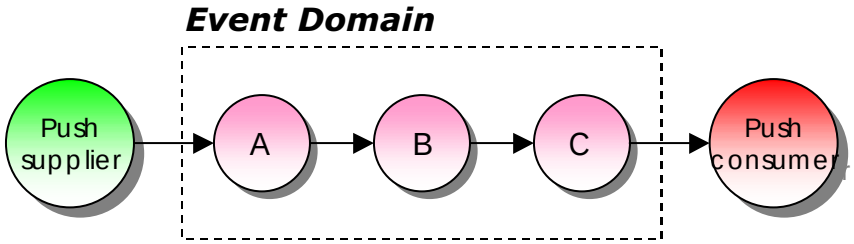

Figure 14 Event Domain and Connected Clients

Creating an Event Domain

Step 1: Create a new (empty) event domain. This requires the following operations:

1. Obtain a reference to the event domain factory. The factory is registered in the Naming Service with the name `EventDomainFactory`.
2. Use the factory's `create_event_domain` method to create the domain. Quality of Service (QoS) and Administrative properties can be specified at this time. (Note, however, that this example does not specify any QoS or Admin property values. See *Using a Domain Factory* on page 138 for an example which sets QoS properties for the domain.)

```
public EventDomain create ()
{
   org.omg.CORBA.Object obj = null;

   try
   {
      obj = orb.resolve_initial_references (“EventDomainFactory”);
   }
   catch (org.omg.CORBA.ORBPackage.InvalidName ex)
   {
      System.err.println (“Failed to resolve Event Domain Factory”);
      System.exit (1);
   }

   EventDomainFactory factory = EventDomainFactoryHelper.narrow (obj);

   Property[] qos = new Property[0];
   Property[] adm = new Property[0];
   org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder ();
   EventDomain domain = null;

   try
   {
      domain = factory.create_event_domain (qos, adm, id);
   }
   catch (UnsupportedQoS ex)
   {
      System.err.println (“UnsupportedQoS”);
      System.exit (1);
   }
   catch (UnsupportedAdmin ex)
   {
      System.err.println (“UnsupportedAdmin”);
      System.exit (1);
   }
```

Step 2: Register the newly-created domain in the root context of the Naming Service. This requires the following operations:

1. Obtain a reference to the Naming Service.
2. Bind the domain into the root context of the Naming Service.

Note that the `register` function is used at several points in the domain creation procedure. It takes an object and the name that the object is to be registered under as parameters. To register the domain, we will pass in the `domain` object (created in Step 1) and the name *MyDomain*.

```
public static void register (org.omg.CORBA.Object object, String name)
{
   org.omg.CORBA.Object obj = null;

   try
   {
      obj = orb.resolve_initial_references (“NameService”);
   }
   catch (org.omg.CORBA.ORBPackage.InvalidName ex)
   {
      System.err.println (“Failed to resolve Name Service”);
      System.exit (1);
   }

   NamingContext root = NamingContextHelper.narrow (obj);

   NameComponent nc[] = new NameComponent[1];
   nc[0] = new NameComponent (name, “Object”);

   try
   {
      root.rebind (nc, object);
      System.out.println (“Placed “ + name + “ in naming context”);
   }
   catch (Exception ex)
   {
      System.err.println (“Failed to bind domain: “ + ex);
      System.exit (1);
   }
}
```

Step 3: Create three Notification Service event channels and add them to the domain. To do this, we use the `setup` function, which performs the following operations:

1. Obtain a reference to the Notification Service event channel factory.
2. Create three new event channels using the factory’s `create_channel` method.
3. Register the event channels in the root context of the Naming Service, using the register function described in Step 2. In this example, we will register the channels under the names *ChannelA*, *ChannelB*, and *ChannelC*.
4. Add the channels to the domain, using the domain’s `add_channel` method.

```
public void setup (EventDomain domain)
{
   org.omg.CORBA.Object obj = null;

   try
   {
```

```
      obj = orb.resolve_initial_references (“NotificationService”);
   }
   catch (org.omg.CORBA.ORBPackage.InvalidName ex)
   {
      System.err.println (“Failed to resolve Notification Service”);
      System.exit (1);
   }

   EventChannelFactory factory = EventChannelFactoryHelper.narrow (obj);
   EventChannel a = null, b = null, c = null;

   try
   {
      Property[] qos = new Property[0];
      Property[] adm = new Property[0];
      org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder ();

      a = factory.create_channel (qos, adm, id);
      b = factory.create_channel (qos, adm, id);
      c = factory.create_channel (qos, adm, id);

      register (a, “ChannelA”);
      register (b, “ChannelB”);
      register (c, “ChannelC”);
   }
   catch (UnsupportedQoS ex)
   {
      System.err.println (“UnsupportedQoS”);
      System.exit (1);
   }
   catch (UnsupportedAdmin ex)
   {
      System.err.println (“UnsupportedAdmin”);
      System.exit (1);
   }

   int idA = domain.add_channel (a);
   int idB = domain.add_channel (b);
   int idC = domain.add_channel (c);
```

Step 4: Set up connections in the new domain, connecting *ChannelA* to *ChannelB* and *ChannelB* to *ChannelC* as shown in *Figure 14* on page 130. This involves two operations:

1. Create the connections. Two individual connections are required, as each connection links two specific channels. Note that the order in which the channels are specified in the creation operation is significant, as connections are *directed* from the first identified channel to the second. The type and style of the connection must also be specified. In this example, the connections are for structured event channels using the push model.
2. Add the new connections to the domain, using the domain’s `add_connection` method.

```
ClientType type = ClientType.STRUCTURED_EVENT;
NotificationStyle style = NotificationStyle.Push;

Connection c1 = new Connection (idA, idB, type, style);
Connection c2 = new Connection (idB, idC, type, style);

try
{
   domain.add_connection (c1);
   domain.add_connection (c2);
```

```
}
catch (Exception ex)
{
   System.err.println ("Failed to created connection: " + ex);
   System.exit (1);
}
```

Step 5: Set *ChannelA* as the default supplier channel and *ChannelC* as the default consumer channel. This ensures that suppliers, by default, will be connected to *ChannelA* whilst consumers, by default, will be connected to *ChannelC*.

```
try
{
   domain.set_default_supplier_channel (idA);
   domain.set_default_consumer_channel (idC);
}
catch (Exception ex)
{
   System.err.println ("Failed to set default channel: " + ex);
   System.exit (1);
}
```

Step 6: Print out some information about the channels, connections, and QoS properties of the newly-created domain. (For further examples of these operations, see page 138 - page 140.) This step is not required, but it allows us to verify that our setup example has worked correctly.

```
int[] chID = myDomain.get_all_channels ();
int[] coID = myDomain.get_all_connections ();

System.out.println ("MyDomain has " + chID.length + " channels");

System.out.println ("Connection information:");

for (int i = 0; i < coID.length; i++)
{
   try
   {
      Connection c = myDomain.get_connection (coID[i]);
      System.out.print (" between channel #" + c.supplier_id);
      System.out.println (" and channel #" + c.consumer_id);
   }
   catch (ConnectionNotFound ex) { }
}

System.out.println ("MyDomain QoS:");

Property[] qos = myDomain.get_qos ();
for (int i = 0; i < qos.length; i++)
{
   System.out.println (" name  = " + qos[i].name);
   System.out.println (" value = " + qos[i].value);
}
```

Connecting a Push Supplier

The following example creates a push supplier and connects it to a channel in the event domain, as shown on the left of *Figure 14* on page 130. Using the Event Domain Service interfaces, the supplier can connect to any of the channels in the domain with a single operation.

The supplier in this example contains methods for publishing events and for disconnecting from the domain. The `publish` method will send 10 events to verify that the domain connections are working correctly.

Step 1: Obtain a reference to the domain, which was registered in the root context of the OpenFusion Naming Service under the name *MyDomain* (see *Creating an Event Domain* on page 130). To do this:

1. Obtain a reference to the root context of the Naming Service.
2. Resolve the name *MyDomain*, which is the name we used to register the domain in the Naming Service (as described previously).

```
public static EventDomain resolve ()
{
   org.omg.CORBA.Object obj = null;

   try
   {
      obj = orb.resolve_initial_references (“NameService”);
   }
   catch (org.omg.CORBA.ORBPackage.InvalidName ex)
   {
      System.err.println (“Failed to resolve Name Service”);
      System.exit (1);
   }

   NamingContext root = NamingContextHelper.narrow (obj);

   NameComponent name[] = new NameComponent[1];
   name[0] = new NameComponent (“MyDomain”, “Object”);

   try
   {
      obj = root.resolve (name);
   }
   catch (NotFound ex)
   {
      return null;
   }
   catch (Exception ex)
   {
      System.err.println (“Failed to resolve MyDomain: “ + ex);
      System.exit (1);
   }

   return EventDomainHelper.narrow (obj);
}
```

Step 2: Connect the supplier to the domain. The supplier’s constructor connects the supplier using the domain’s `connect_structured_push_supplier` method. In this example, we will connect to the domain’s default supplier channel.

```
public class Supplier implements StructuredPushSupplierOperations
{
   public static void main (String[] args)
   {
      //In order to make examples easier to run, for Orbacus-40 set a POAName
      if (com.prismt.openfusion.Version.getORB().
            toUpperCase().startsWith(“ORBACUS-4”))
      {
         ObjectAdapter.setPOAName(“OpenFusion.EventDomainSupplier”);
```

```
      }

      orb = ObjectAdapter.init (args);
      Setup.orb = orb;

      System.out.println (“Connecting”);
      Supplier supplier = new Supplier ();

      System.out.println (“Supplying”);
      supplier.publish ();

      System.out.println (“Disconnecting”);
      supplier.disconnect ();

      System.out.println (“Success”);

      ObjectAdapter.shutdown();
   }

   public Supplier ()
   {
      org.omg.CORBA.Object ref = ObjectAdapter.createTransient (this);
      EventDomain domain = Setup.resolve ();

      if (domain == null)
      {
         System.err.println (“MyDomain not found”);
         System.exit (1);
      }

      try
      {
         StructuredPushSupplier supplier;
         supplier = StructuredPushSupplierHelper.narrow (ref);
         proxy = domain.connect_structured_push_supplier (supplier);
         ObjectAdapter.ready (false);
      }
      catch (ChannelNotFound ex)
      {
         System.err.println (“ChannelNotFound”);
         System.exit (1);
      }
   }

   public void disconnect_structured_push_supplier ()
   {
   }

   public void subscription_change (EventType[] added, EventType[] removed)
      throws InvalidEventType
   {
      System.out.println (“Added types:”);
      for (int i = 0; i < added.length; i++)
      {
         System.out.println (added[i]);
      }
      System.out.println (“Removed types:”);
      for (int i = 0; i < removed.length; i++)
      {
         System.out.println (removed[i]);
      }
   }

   public void publish ()
   {
      StructuredEvent event = new StructuredEvent ();
      event.header = new EventHeader ();
      event.header.fixed_header = new FixedEventHeader ();
```

```
      event.header.fixed_header.event_type = new EventType (“”, “”);
      event.header.fixed_header.event_name = “”;
      event.header.variable_header = new Property[0];
      event.filterable_data = new Property[0];

      for (int i = 0; i < 10; i++)
      {
         try
         {
            event.remainder_of_body = orb.create_any ();
            event.remainder_of_body.insert_long (i);
            proxy.push_structured_event (event);
         }
         catch (org.omg.CosEventComm.Disconnected ex)
         {
            System.out.println (“Disconnected”);
            System.exit (0);
         }
      }
   }

   public void disconnect ()
   {
      proxy.disconnect_structured_push_consumer ();
   }

   private static org.omg.CORBA.ORB orb = null;
   private StructuredProxyPushConsumer proxy = null;
}
```

Connecting a Push Consumer

The following example creates a push consumer and connects it to a channel in the event domain, as shown on the right of *Figure 14* on page 130. Using the Event Domain Service interfaces, the consumer can connect to any of the channels in the domain with a single operation.

The consumer prints out the 10 events sent by the supplier created in *Connecting a Push Supplier* on page 133. The events have passed through channels *A*, *B*, and *C* as shown in Figure 14, *Event Domain and Connected Clients* on page 130.

Step 1: Obtain a reference to the domain. This is as described in Step 1 of *Connecting a Push Supplier* on page 133.

Step 2: Connect the consumer to the domain. The consumer’s constructor connects the consumer using the domain’s `connect_structured_push_consumer` method. In this example, we will connect to the domain’s default consumer channel.

```
public class Consumer implements StructuredPushConsumerOperations
{
   public static void main (String[] args)
      throws java.io.IOException
   {
      //In order to make examples easier to run, for Orbacus-40 set a POAName
      if (com.prismt.openfusion.Version.getORB().
            toUpperCase().startsWith(“ORBACUS-4”))
      {
         ObjectAdapter.setPOAName(“OpenFusion.EventDomainConsumer”);
      }

      orb = ObjectAdapter.init (args);
```

```
      Setup.orb = orb;

      Consumer consumer = new Consumer ();

      com.prismt.orb.ObjectAdapter.ready (false);

      System.out.println ("Consumer Ready. Press Return to quit");

      System.in.read();
      ObjectAdapter.shutdown();
   }

   public Consumer ()
   {
      org.omg.CORBA.Object ref = ObjectAdapter.createTransient (this);
      EventDomain domain = Setup.resolve ();

      if (domain == null)
      {
         System.err.println ("MyDomain not found");
         System.exit (1);
      }

      try
      {
         StructuredPushConsumer consumer;
         consumer = StructuredPushConsumerHelper.narrow (ref);
         proxy = domain.connect_structured_push_consumer (consumer);
      }
      catch (ChannelNotFound ex)
      {
         System.err.println ("ChannelNotFound");
         System.exit (1);
      }
   }

   public void disconnect_structured_push_consumer ()
   {
      System.out.println ("disconnect_structured_push_consumer");
      System.exit (0); // stop
   }

   public void offer_change (EventType[] added, EventType[] removed)
      throws InvalidEventType
   {
      System.out.println ("Added types:");
      for (int i = 0; i < added.length; i++)
      {
         System.out.println (added[i]);
      }
      System.out.println ("Removed types:");
      for (int i = 0; i < removed.length; i++)
      {
         System.out.println (removed[i]);
      }
   }

   public void push_structured_event (StructuredEvent event)
   {
      System.out.println (event.remainder_of_body);
   }

   private static org.omg.CORBA.ORB orb = null;
   private StructuredProxyPushSupplier proxy = null;
}
```

10.2 Managing Untyped Event Domains

An event domain is a collection manager for the channels and connections that make up the domain. The untyped event domain interfaces are defined in the `org.omg.CosEventDomainAdmin` package. These interfaces can be used to manage untyped event channels, as shown in the following examples. See Section 10.3, *Managing Typed Event Domains* on page 146, for examples of managing typed event channels.

Using a Domain Factory

A domain factory is used to create new event domains. Each domain created by a domain factory is identified by an integer, which is unique within the scope of that factory. The factory can manage the collection of the domains it has created.

The example below shows how to create an event domain with QoS properties set, as follows:

1. Create an array of properties and populate it with any required QoS properties.
2. Use the `create_event_domain` method of the event domain factory to create the domain.

```
Property[] qos = new Property[2];

qos[0] = new Property (CycleDetection.value, orb.create_any ());
qos[1] = new Property (DiamondDetection.value, orb.create_any ());
qos[0].value.insert_short (ForbidCycles.value);
qos[1].value.insert_short (ForbidDiamonds.value);

Property[] adm = new Property[0];
org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder ();
EventDomain domain = null;

try
{
   domain = factory.create_event_domain (qos, adm, id);
}
catch (UnsupportedQoS ex)
{
   System.err.println (“UnsupportedQoS”);
   System.exit (1);
}
catch (UnsupportedAdmin ex)
{
   System.err.println (“UnsupportedAdmin”);
   System.exit (1);
}
```

Listing the Quality of Service Properties

This example below prints the QoS properties of all domains that have been created by a factory, as follows:

1. The domain factory's `get_all_domains` method returns the identifiers of the domains in the collection.
2. The factory's `get_event_domain` method returns a specific domain from the collection.
3. The domain's `get_qos` method returns the collection of QoS properties for the domain.

```
int[] ids = factory.get_all_domains ();
for (int i = 0; i < ids.length; i++)
{
   try
   {
      EventDomain domain = factory.get_event_domain (ids[i]);
      Property[] qos = domain.get_qos ();
      System.out.println ("QoS for domain #" + ids[i]);
      for (int j = 0; j < qos.length; j++)
      {
         System.out.println (" name  = " + qos[j].name);
         System.out.println (" value = " + qos[j].value);
      }
   }
   catch (DomainNotFound ex) { } // ignore
}
```

Destroying a Domain

The `destroy` operation removes a domain from a factory collection. In addition, all existing connections between the channels in the domain are also removed. Destroying a domain has the same effect as invoking the `remove_connection` operation on each individual connection in the domain.

Managing Channels

The following channel-management operations are provided:

- `add_channel`
- `get_all_channels`
- `get_channel`
- `remove_channel`

The `add_channel` operation is illustrated in Step 3 of *Creating an Event Domain* on page 131. The other three operations are illustrated in the following example. This example removes all channels in a domain that have event reliability set to *best effort*. Note that removing a channel automatically removes all existing connections to it.

```
int[] ids = domain.get_all_channels ();
for (int i = 0; i < ids.length ; i++)
{
   try
   {
      EventChannel channel = domain.get_channel (ids[i]);
```

```
      Property[] qos = channel.get_qos ();
      for (int j = 0; j < qos.length; j++)
      {
         if (qos[j].name.equals(EventReliability.value))
         {
            if (qos[j].value.extract_short() == BestEffort.value)
            {
               domain.remove_channel (ids[i]);
            }
         }
      }
   }
   catch (ChannelNotFound ex) { } // ignore
}
```

Managing Connections

The following operations are provided to allow connection management:

- `add_connection`
- `get_all_connections`
- `get_connection`
- `remove_connection`

The Event Domain Service uses the `Connection` data structure shown in *Table 12* to describe the connections in an event domain.

Table 12 Connection Data Structure

`SupplierChannel`
`ConsumerChannel`
`ClientType`
`NotificationStyle`

A connection is directed so that the `SupplierChannel` is the source of events while the `ConsumerChannel` is the target. The `ClientType` may be `untyped` (`ANY_EVENT`), `structured` (`STRUCTURED_EVENT`), or a `sequence` (`SEQUENCE_EVENT`). The `NotificationStyle` can be either *push* or *pull*.

The `add_connection` operation is illustrated in Step 4 of Creating an Event Domain on page 132. The other three operations are illustrated in the following example. This example removes all connections with a client type of `sequence` from the domain.

```
int[] ids = domain.get_all_connections ();
for (int i = 0; i < ids.length ; i++)
{
   try
   {
      Connection c = domain.get_connection (ids[i]);
```

```
      if (c.ctype == ClientType.SEQUENCE_EVENT)
      {
         domain.remove_connection (ids[i]);
      }
   }
   catch (ConnectionNotFound ex) { } // ignore
}
```

The following situations can cause problems in domain management and should be avoided:

- Connections may be made between channels without using the `add_connection` operation of the Event Domain Service. Applications could manually add such connections using the standard operations of the Notification Service. Such connections will *not* be visible to the Event Domain Service.
- It is possible to add the same event channel to a domain more than once. Event channels are identified only by number, since it is not generally possible to reliably compare CORBA object references.

Connecting Clients

An untyped event domain supports operations for connecting consumers and suppliers to event channels. These operations can connect to the default supplier and consumer channels, or to a specific channel by explicitly specifying the channel's unique identifier in the `connect` operation.

The default supplier channel is defined with the domain's `set_default_supplier_channel` operation. The default consumer channel is defined with the `set_default_consumer_channel` operation. Step 5 of Creating an Event Domain on page 133 has an example of using these methods.

Note that if a default supplier or consumer channel is not defined, then the first channel added to the domain is used as the default.

There are different operations for connecting suppliers and consumers for each client type (`untyped`, `structured`, and `sequence`) and each communication model (`push` and `pull`). For example:

- `connect_push_supplier`
- `connect_push_consumer`
- `connect_structured_push_supplier`
- `connect_structured_push_consumer`

The full list of operations is given in Section 11.1, *Interfaces*, on page 151.

Each connect operation performs the following steps:

Step 1: Obtains the supplier or consumer admin object from the target channel. A `ChannelNotFound` exception is raised if the target channel does not exist.

Step 2: Obtains a proxy from the admin object according to the client type (`untyped`, `structured` or `sequence`) and communication model (`push` or `pull`). An `IMP_LIMIT` system exception is raised if the admin object raises an `AdminLimitExceeded` exception.

Step 3: Connects the client to the newly created proxy object. An `INTERNAL` system exception is raised when the proxy raises an `AlreadyConnected` or a `TypeError` exception, since this is not supposed to happen.

All of this is accomplished by a single line of code, as illustrated by the examples in *Connecting a Push Supplier* on page 133 and *Connecting a Push Consumer* on page 136.

Topology Management

The Event Domain Service supports several operations for topology management. The key operations provide information about two key topographical features which may occur in the domain: cycles and diamonds.

Cycles

If the `CycleDetection` QoS property has a value of `AuthorizeCycles`, a domain may contain cycles.

Consider the event domain shown in *Figure 15*. This domain has three cycles. The `get_cycles` operation returns a sequence, which in turn contains a sequence of channel identifiers. The return value is therefore an array of arrays as illustrated to the right of *Figure 15*.

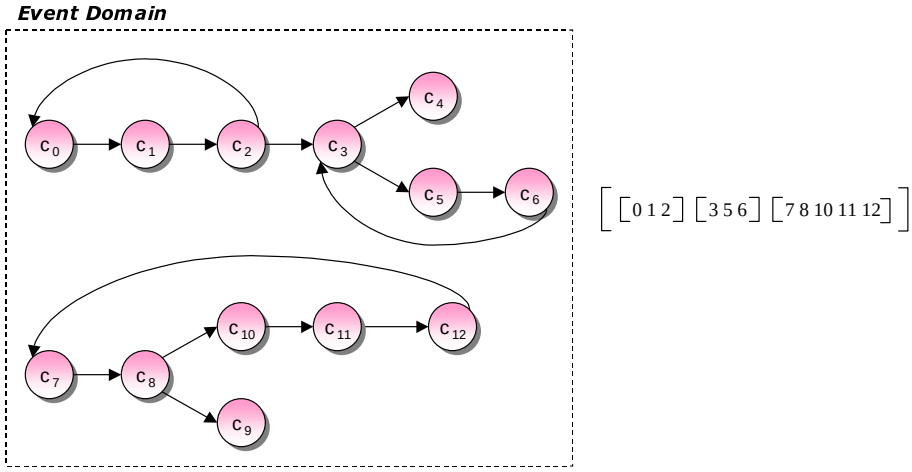

Figure 15 Domain Containing Three Cycles

Note that channels *4* and *9* are not part of any cycles, and therefore do not appear in the returned sequence.

The following example uses the `get_cycles` operation to print out all the cycles in a domain:

```
int[][] cycles = domain.get_cycles ();
for (int i = 0; i < cycles.length; i++)
{
   System.out.print (“Cycle: “);
   for (int j = 0; j < cycles[i].length; j++)
   {
      System.out.print (cycles[i][j] + “ “);
   }
   System.out.println ();
}
```

The output from running this example on the domain shown in *Figure 15* is:

```
Cycle: 0 1 2
Cycle: 3 5 6
Cycle: 7 8 10 11 12
```

i The order of channel identifiers in the return sequences may not be precisely as indicated in the above graph for the `get_cycles` operation. Although the sequence will be ordered correctly, it may start with any channel in the cycle.

Diamonds

If the `DiamondDetection` QoS property has a value of `AuthorizeDiamonds`, a domain may contain diamonds.

Consider the event domain shown in *Figure 16*. This domain has three diamonds, where one of the diamonds has three edges. The `get_diamonds` operation returns a sequence of diamonds. Each diamond is a sequence of *routes*. A route is a sequence with the identifiers of all channels that participate in a diamond path. The return value is thus an array of integer arrays as illustrated to the right of *Figure 16*.

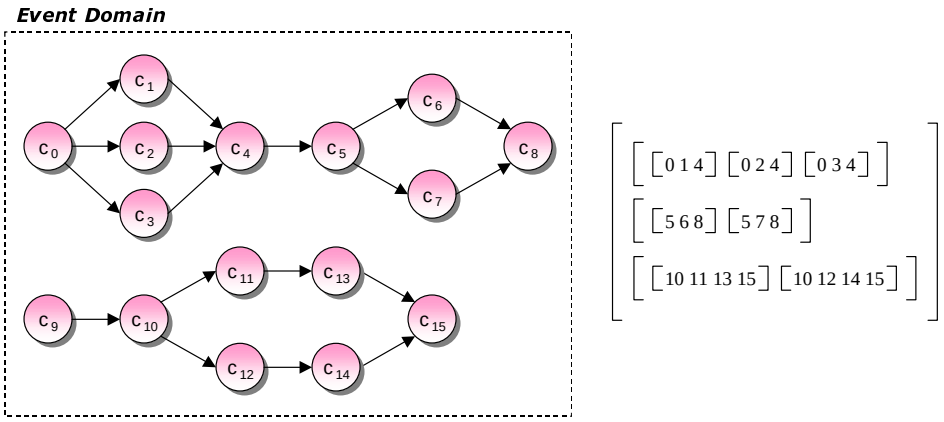

Figure 16 Domain Containing Three Diamonds

Note that channel *9* is not a part of any diamonds, and therefore does not appear in the returned sequence.

The following example uses the `get_diamonds` operation to print out all the diamonds in a domain:

```
int[][][] diamonds = domain.get_diamonds ();
for (int i = 0; i < diamonds.length; i++)
{
   System.out.println (“Paths in diamond #” + i);
   for (int j = 0; j < diamonds[i].length; j++)
   {
      System.out.print (“ path #” + j + “: “);
      for (int k = 0; k < diamonds[i][j].length; k++)
      {
         System.out.print (diamonds[i][j][k] + “ “);
      }
      System.out.println ();
   }
}
```

The output from running this example on the domain shown in *Figure 16* is:

```
Paths in diamond #0:
 path #0: 0 1 4
 path #1: 0 2 4
 path #2: 0 3 4

Paths in diamond #1:
 path #0: 5 6 8
 path #1: 5 7 8

Paths in diamond #2:
 path #0: 10 11 13 15
 path #1: 10 12 14 15
```

Channels

The Event Domain Service supports the following additional operations for obtaining information about the topology of an event domain:

- *get_offer_channels* - Returns an array of channel identifiers for all channels upstream from the specified target channel
- *get_subscription_channels* - returns an array of channel identifiers for all channels downstream from the specified target channel

Disabling Event Type Propagation

The Event Domain Service also supports an additional QoS setting, `EventTypesEnabled`, to control event type propagation in an event domain. An event type change can cause a large number of callbacks in a network of many channels, so applications may disable event type propagation for performance reasons. The default is for event type propagation to be enabled.

Event type subscription changes will affect all channels upstream from the initiating consumer, and event type offer changes will affect all channels downstream from the initiating supplier.

Event type information will propagate through a domain as follows:

1. A consumer connected to an event channel changes its subscribed types, either by calling the proxy's `subscription_change` operation or by manipulating the event types associated with a Notification Service filter constraint.
2. The proxy notifies the channel about this change.

3. The channel informs all connected suppliers (by invoking their `subscription_change` operation) when the newly added or removed event type modifies the event type aggregate at the channel.
4. The event type information is propagated back through the event channels in the domain.

i Event type callbacks will never endlessly propagate through the event system when there is a cycle, because a channel will only issue event type callbacks when the aggregate of subscribed or offered types changes.

10.3 Managing Typed Event Domains

Typed event domains are collections of typed event channels and connections. Typed event domain interfaces are defined in the `org.omg.CosTypedEventDomainAdmin` package. The basic functionality is the same as that of the untyped event domain, described in 10.2, *Managing Untyped Event Domains* on page 138. Additional operations for connecting typed clients are described below.

Using a Typed Event Domain Factory

The Typed Event Domain Factory supports the same operations as the untyped event domain factory. The factory is resolved by using the name `TypedEventDomainFactory`, as shown in the example below:

```
org.omg.CORBA.Object obj = null;

try
{
   obj = orb.resolve_initial_references (“TypedEventDomainFactory”);
}
catch (org.omg.CORBA.ORBPackage.InvalidName ex)
{
   System.err.println (“Failed to resolve Typed Event Domain Factory”);
   System.exit (1);
}

TypedEventDomainFactory factory;
factory = TypedEventDomainFactoryHelper.narrow (obj);
```

Managing Typed Channels

Typed event channels are added to a typed event domain using the `add_typed_channel` operation.

It is possible to add untyped event channels to a typed event domain, since the `TypedEventDomain` interface inherits from the `EventDomain` interface. Untyped event channels are added using the `add_channel` operation.

Managing Typed Connections

The `TypedConnection` data structure shown in *Table 13* describes the connections in a typed event domain. Compare this structure with Table 12, *Connection Data Structure*: the typed event model does not support *client type* and instead uses a *repository identifier* (the `Key` field).

Table 13 TypedConnection Data Structure

`SupplierChannel`
`ConsumerChannel`
`Key`
`NotificationStyle`

The following example shows how to create a typed connection between two channels. The channels have the identifiers `idA` and `idB` (assumed to be initialized elsewhere in the code).

```
try
{
   NotificationStyle style = NotificationStyle.Push;
   String id = AccountObserverHelper.id();
   TypedConnection c = new TypedConnection (idA, idB, id, style);
   domain.add_typed_connection (c);
}
catch (ChannelNotFound ex)
{
   System.err.println (“ChannelNotFound”);
   System.exit (1);
}
catch (org.omg.CosEventChannelAdmin.TypeError ex)
{
   System.err.println (“TypeError”);
   System.exit (1);
}
catch (AlreadyExists ex)
{
   System.err.println (“AlreadyExists”);
   System.exit (1);
}
catch (CycleCreationForbidden ex)
{
   System.err.println (“CycleCreationForbidden”);
   System.exit (1);
}
catch (DiamondCreationForbidden ex)
{
   System.err.println (“DiamondCreationForbidden”);
   System.exit (1);
}
```

Refer to the *Typed Notification Service* documentation for information about the `AccountObserver` interface.

The Event Domain Service contains a significant limitation, caused by an error in the OMG Event and Notification Service specifications. An inheritance flaw in the specifications makes it impossible to use the Event Domain Service to create a connection between a typed event channel and an untyped event channel. This also means that the `get_channel` operation of the untyped event domain cannot return a typed event channel. Clients should use the `get_typed_channel` operation to retrieve a typed event channel.

Connecting Typed Clients

A typed event domain supports operations for connecting typed suppliers and consumers to typed event channels. These operations can connect to the default supplier and consumer channels, or to a specific channel by explicitly specifying the channel's unique identifier in the `connect` operation.

The default typed supplier event channel is defined with the `set_default_typed_consumer_channel` operation. The default typed consumer channel is defined with the `set_default_typed_supplier_channel` operation. These operations are used identically to the equivalent operations provided for untyped domains (see Step 5 of Creating an Event Domain on page 133 for an example.)

Note that if a default supplier or consumer channel is not defined, then the first channel added to the domain is used as the default.

There are different operations for connecting suppliers and consumers for each communication model (`push` and `pull`). For example:

- `connect_typed_push_supplier`
- `connect_typed_pull_supplier`

The full list of operations is given in Section 11.1, *Interfaces* on page 151.

The client application must specify the repository identifier of the interface to be used for typed event communication (in addition to the arguments supplied with the connect operations as with an untyped event domain). The format of this string is the same as that used for the `Key` field in the `TypedConnection` data structure.

10.4 Log Domains

A log domain is functionally similar to an event domain, except that it supports the management of log objects and typed log objects. The `EventLogDomain` interface inherits from the `TypedEventDomain` interface, so a log domain supports all the operations described in the previous sections.

The log domain factory is functionally identical to the event domain factory and typed event domain factory, previously described. This factory is resolved using the name `EventLogDomainFactory`. It supports the creation and collection management of log domains.

Log domains support the type-safe addition and retrieval of logs and typed logs through the following operations:

- `add_log`
- `add_typed_log`
- `get_log`
- `get_typed_log`.

i Note: typed and untyped logs both are handled by the same classes (`EventLogDomain` and `EventLogDomainFactory`). There are no separate classes for typed logs.

11 API Definitions

This chapter describes the main Event Domain interfaces. The complete IDL API is provided elsewhere as part of the product distribution.

11.1 Interfaces

The Event Domain Service interfaces are listed in *Table 14*:

Table 14 Event Domain Service Interfaces

Interface	Description
`EventDomain`	An event domain for federating and managing untyped event channels, and for connecting event suppliers and consumers to event channels.
`EventDomainFactory`	A factory for creating and managing untyped event domains.
`EventLogDomain`	An event domain for managing logs and typed logs.
`EventLogDomainFactory`	A factory for managing logs and typed logs.
`TypedEventDomain`	An event domain for managing typed event channels.
`TypedEventDomainFactory`	A factory for creating and managing typed event channels.

The `EventDomain` interfaces, as shown in *Figure 17*, support operations for managing untyped event channels and connections within a domain, as well as for connecting consumers and suppliers to an event channel within the domain. In addition, the interfaces have operations for domain topography management: for obtaining upstream and downstream channel information, and for listing the cycles and diamonds within a domain.

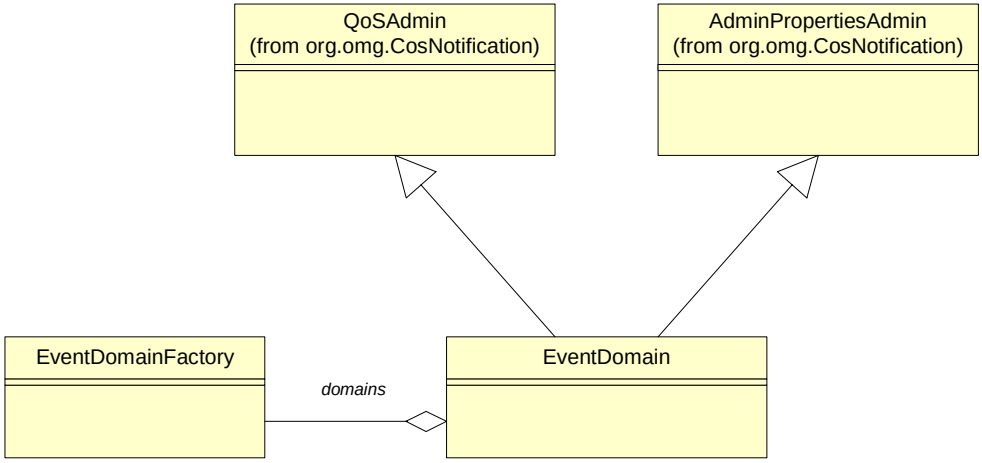

Figure 17 CosEventDomainAdmin Module Interfaces

The `TypedEventDomain` interfaces, as shown in *Figure 18*, inherit from the `EventDomain` interfaces and include additional operations for the connection of typed clients to a typed event channel. These operations have an additional argument to those of the corresponding untyped operations: the *repository identifier* that specifies the interface to be used for typed event communication.

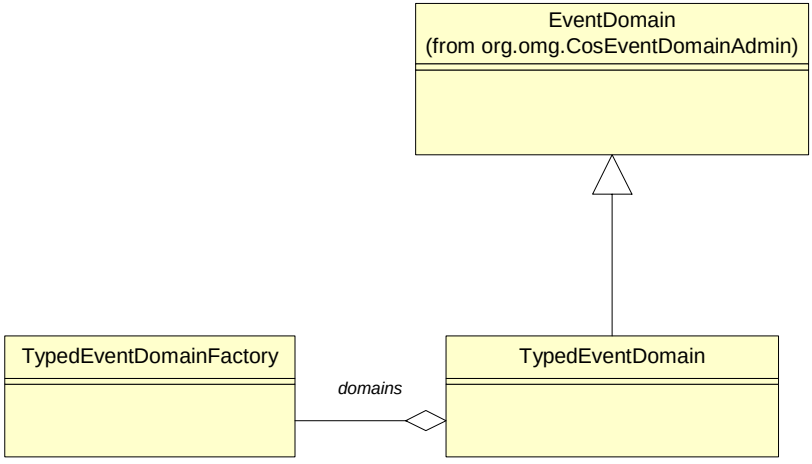

Figure 18 CosTypedEventDomainAdmin Module Interfaces

A log domain is very similar to an event domain, since a log is functionally equivalent to an event channel. Inheritance means that Log domains require very few additional operations to support their management. The `EventLogDomain` interfaces are shown in *Figure 19*. A log domain, like an event domain, supports only notification style log objects. The only other operations in the `EventLogDomain` interfaces are used for the type-safe addition and retrieval of typed log objects.

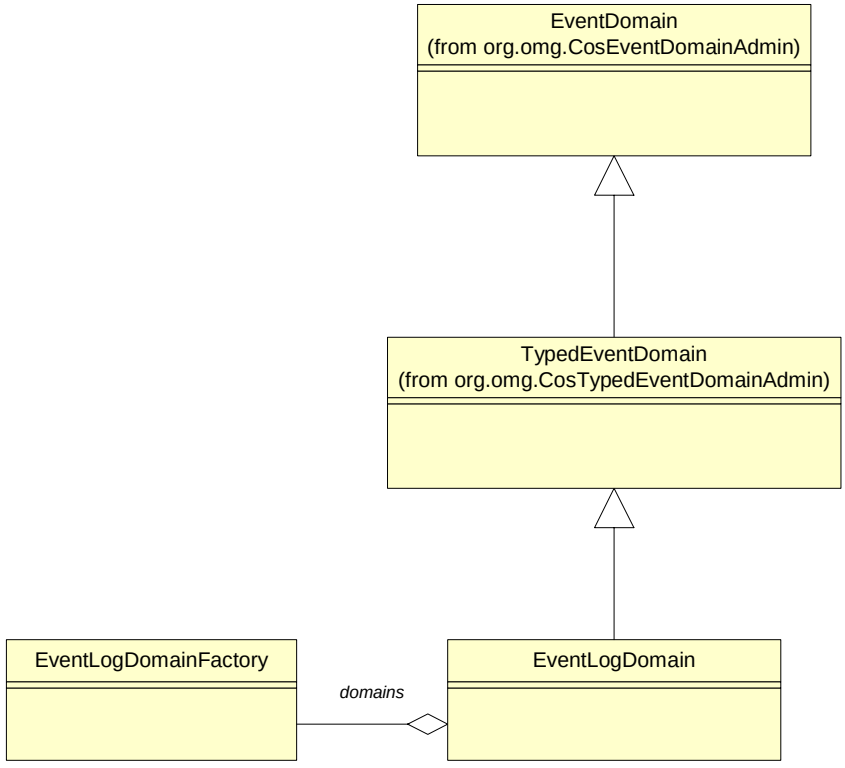

Figure 19 DsLogDomainAdmin Module Interfaces

EventDomain

This is the main interface for federating untyped event channels, and for connecting suppliers and consumers to an event channel.

Operations

add_channel

Adds an untyped event channel to a domain.

add_connection

Connects two event channels in a domain. If either channel does not exist, a `ChannelNotFound` exception is raised

If the two channels are already connected, an `AlreadyExists` exception is raised. This exception is also raised if a channel is being connected to itself (that is, the same channel is specified at both ends of the connection).

If the `CycleDetection` QoS property is set to `ForbidCycle`, and the creation of the requested connection would result in a cycle configuration, a `CycleCreationForbidden` exception is raised.

If the `DiamondDetection` QoS property is set to `ForbidDiamond`, and the creation of the requested connection would result in a diamond configuration, a `DiamondCreationForbidden` exception is raised.

connect_pull_consumer

Connects a pull consumer to the default consumer channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_pull_consumer_with_id

Connects a pull consumer to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_pull_supplier

Connects a pull supplier to the default supplier channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_pull_supplier_with_id

Connects a pull supplier to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_push_consumer

Connects a push consumer to the default consumer channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_push_consumer_with_id

Connects a push consumer to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_push_supplier

Connects a push supplier to the default supplier channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_push_supplier_with_id

Connects a push supplier to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_sequence_pull_consumer

Connects a sequence pull consumer to the default consumer channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_sequence_pull_consumer_with_id

Connects a sequence pull consumer to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_sequence_pull_supplier

Connects a sequence pull supplier to the default supplier channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_sequence_pull_supplier_with_id

Connects a sequence pull supplier to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_sequence_push_consumer

Connects a sequence push consumer to the default consumer channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_sequence_push_consumer_with_id

Connects a sequence push consumer to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_sequence_push_supplier

Connects a sequence push supplier to the default supplier channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_sequence_push_supplier_with_id

Connects a sequence push supplier to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_structured_pull_consumer

Connects a structured pull consumer to the default consumer channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_structured_pull_consumer_with_id

Connects a structured pull consumer to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_structured_pull_supplier

Connects a structured pull supplier to the default supplier channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_structured_pull_supplier_with_id

Connects a structured pull supplier to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_structured_push_consumer

Connects a structured push consumer to the default consumer channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_structured_push_consumer_with_id

Connects a structured push consumer to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

connect_structured_push_supplier

Connects a structured push supplier to the default supplier channel of a target domain. If no channels are found, a `ChannelNotFound` exception will be raised.

connect_structured_push_supplier_with_id

Connects a structured push supplier to a specified channel of a target domain. A `ChannelNotFound` exception will be raised if the channel does not exist.

destroy

Removes a domain from a factory collection. This will also remove any existing connections between channels in the domain.

get_all_channels

Returns a sequence of all the unique identifiers corresponding to all the existing channels in a domain.

get_all_connections

Returns a sequence of the unique identifiers corresponding to all the existing connections in a domain.

get_channel

Uses the unique channel identifier to return an object reference to a specific channel in a domain. A `ChannelNotFound` exception will be raised if no channel corresponding to the specified identifier exists.

get_connection

Uses a connection's unique identifier to return the `Connection` data structure for that connection (described in *Managing Connections* on page 140). A `ConnectionNotFound` exception will be raised if no connection corresponding to the identifier exists or if the connection is a typed connection.

get_cycles

Returns a sequence of all the cycles in a domain.

get_diamonds

Returns a sequence of all the diamonds in a domain.

get_offer_channels

Returns a list of all channels that exist upstream of a specified channel in a domain. A `ChannelNotFound` exception will be raised if the specified channel does not exist.

get_subscription_channels

Returns a list of all channels that exist downstream of a specified channel in a domain. A `ChannelNotFound` exception will be raised if the specified channel does not exist.

remove_channel

Removes a channel from a domain. This also removes all existing connections to the channel. A `ChannelNotFound` exception will be raised if the specified channel does not exist.

remove_connection

Removes a connection between two specified channels in a domain. A `ConnectionNotFound` exception will be raised if the specified connection does not exist.

set_default_consumer_channel

Used to define the default consumer channel for a domain. A `ChannelNotFound` exception will be raised if the specified channel does not exist.

set_default_supplier_channel

Used to define the default supplier channel for a domain. A `ChannelNotFound` exception will be raised if the specified channel does not exist.

EventDomainFactory

A factory interface for creating and managing event domains.

Operations

create_event_domain

Creates a new instance of an event domain. Takes the following parameters:

- A list of name-value pairs that specify the initial QoS properties for the new domain. If no implementation of the `EventDomain` interface exists that can support all of the requested QoS property settings, an `UnsupportedQoS` exception is raised.
- A list of name-value pairs that specify the initial administration properties for the new domain. If no implementation of the `EventDomain` interface exists that can support the requested administration properties, an `UnsupportedAdmin` exception is raised.

get_all_domains

Returns a list of all the domains that have been created by the factory.

get_event_domain

Uses the unique domain identifier to return an object reference to a specific domain that has been created by this factory. A `DomainNotFound` exception will be raised if no domain corresponding to the specified identifier exists.

EventLogDomain

An event domain interface for managing logs and typed logs.

Operations

add_log

Adds an untyped Notification log channel to the domain.

add_typed_log

Adds a typed Notification log channel to the domain.

get_log

Uses the unique log channel identifier to return an object reference to a specific untyped log channel in the domain. A `ChannelNotFound` exception will be raised if no log corresponding to the specified identifier exists.

get_typed_log

Uses the unique log channel identifier to return an object reference to a specific typed log in the domain. A `ChannelNotFound` exception will be raised if no log corresponding to the specified identifier exists.

EventLogDomainFactory

A factory interface for managing logs and typed logs.

Operations

create_event_log_domain

Creates a new instance of an event log domain. Takes the following parameters:

- A list of name-value pairs that specify the initial QoS properties for the new domain. If no implementation of the `EventLogDomain` interface exists that can support all of the requested QoS property settings, an `UnsupportedQoS` exception is raised.
- A list of name-value pairs that specify the initial administration properties for the new domain. If no implementation of the `EventLogDomain` interface exists that can support the requested administration properties, an `UnsupportedAdmin` exception is raised.

get_all_event_log_domains

Returns a list of all the event log domains that have been created by the factory.

get_event_log_domain

Uses the unique domain identifier to return an object reference to an event log domain that has been created by this factory. A `DomainNotFound` exception will be raised if no domain corresponding to the specified identifier exists.

TypedEventDomain

An interface for managing typed event channels.

Operations

add_typed_channel

Adds a typed event channel to a domain.

add_typed_connection

Forms a typed connection between two typed event channels in the domain. If either channel does not exist, a `ChannelNotFound` exception is raised.

If the two channels are already connected, an `AlreadyExists` exception is raised. This exception is also raised if a channel is being connected to itself (that is, the same channel is specified at both ends of the connection).

If either of the two channels is not a typed event channel, a `TypeError` exception is raised.

If the `CycleDetection` QoS property is set to `ForbidCycle`, and the creation of the requested connection would result in a cycle, a `CycleCreationForbidden` exception is raised.

If the `DiamondDetection` QoS property is set to `ForbidDiamond`, and the creation of the requested connection would result in a diamond, a `DiamondCreationForbidden` exception is raised.

connect_typed_pull_consumer

Connects a typed pull consumer to the domain's default typed consumer channel. If the target domain contains no typed channels, a `ChannelNotFound` exception is raised.

If the default channel for typed consumers is not capable of creating a typed proxy supplier that supports the specified interface, an `InterfaceNotSupported` exception is raised.

connect_typed_pull_consumer_with_id

Connects a typed pull consumer to a specified channel in the target domain. If the specified channel does not exist, a `ChannelNotFound` exception is raised.

If the specified channel is not capable of creating a typed proxy supplier that supports the specified interface, an `InterfaceNotSupported` exception is raised.

connect_typed_pull_supplier

Connects a typed pull supplier to the domain's default typed supplier channel. If the target domain contains no typed channels, a `ChannelNotFound` exception is raised.

If the default channel for typed suppliers does not support the ability to pull typed events, a `NoSuchImplementation` exception is raised.

connect_typed_pull_supplier_with_id

Connects a typed pull supplier to a specified channel in the target domain. If the specified channel does not exist, a `ChannelNotFound` exception is raised.

If the specified channel does not support the ability to pull typed events, a `NoSuchImplementation` exception is raised.

If the typed supplier does not support the specified interface, a `TypeError` exception is raised.

connect_typed_push_consumer

Connects a typed push consumer to the domain's default typed consumer channel. If the target domain contains no typed channels, a `ChannelNotFound` exception is raised.

If the default channel for typed consumers does not support the ability to push typed events, a `NoSuchImplementation` exception is raised.

connect_typed_push_consumer_with_id

Connects a typed push consumer to a specified channel in the target domain. If the specified channel does not exist, a `ChannelNotFound` exception is raised.

If the specified channel does not support the ability to push typed events, a `NoSuchImplementation` exception is raised.

If the typed consumer does not support the specified interface, then the `TypeError` exception is raised.

connect_typed_push_supplier

Connects a typed push supplier to the domain's default typed supplier channel. If the target domain contains no typed channels, a `ChannelNotFound` exception is raised.

If the default channel for typed suppliers is not capable of creating a typed proxy consumer that supports the specified interface, an `InterfaceNotSupported` exception is raised.

connect_typed_push_supplier_with_id

Connects a typed push supplier to a specified channel in the target domain. If the specified channel does not exist, a `ChannelNotFound` exception is raised.

If the specified channel is not capable of creating a typed proxy consumer that supports the specified interface, an `InterfaceNotSupported` exception is raised.

get_typed_channel

Uses a unique identifier to return the object reference of a typed channel in the target domain. A `ChannelNotFound` exception will be raised if no channel corresponding to the specified identifier exists.

get_typed_connection

Uses a unique identifier to return the object reference of a typed connection in the target domain. A `ConnectionNotFound` exception will be raised if no connection corresponding to the specified identifier exists, or if the connection is not a typed connection.

set_default_typed_consumer_channel

Used to define the default typed consumer channel for a domain. A `ChannelNotFound` exception will be raised if the specified channel does not exist.

set_default_typed_supplier_channel

Used to define the default typed supplier channel for a domain. A `ChannelNotFound` exception will be raised if the specified channel does not exist.

TypedEventDomainFactory

A factory interface for creating and managing typed event domains.

Operations

create_typed_event_domain

Creates a new instance of a typed event domain. Takes the following parameters:

- A list of name-value pairs that specify the initial QoS properties for the new domain. If no implementation of the `TypedEventDomain` interface exists that can support all of the requested QoS property settings, an `UnsupportedQoS` exception is raised.
- A list of name-value pairs that specify the initial administration properties for the new domain. If no implementation of the `TypedEventDomain` interface exists that can support the requested administration properties, an `UnsupportedAdmin` exception is raised.

get_all_typed_domains

Returns a list of all the typed event domains that have been created by the factory.

get_typed_event_domain

Uses a unique identifier to return the object reference to a typed event domain that has been created by this factory. A `DomainNotFound` exception will be raised if no typed event domain corresponding to the specified identifier exists.

12 Supplemental Information

12.1 Quality of Service Properties

The OpenFusion implementation of the Event Domain Service currently supports three different QoS properties, as described in *Table 15*.

Table 15 Event Domain Service QoS Properties

Property	Description
CycleDetection	When this is set to `ForbidCycles`, the domain raises a `CycleCreationForbidden` exception when attempting to add a connection that will form a cycle. When this is set to `AuthorizeCycles`, the creation of cycles will be allowed and will not be flagged in any way. The default is `AuthorizeCycles`.
DiamondDetection	When this is set to `ForbidDiamonds`, the domain raises a `DiamondCreationForbidden` exception when attempting to add a connection that will form a diamond. When this is set to `AuthorizeDiamonds`, the creation of diamonds will be allowed and will not be flagged in any way. The default is `AuthorizeDiamonds`.
EventTypesEnabled	When this is set to `true`, the domain will enable propagation of event type information. This means that `get_offered_types` and `get_subscription_types` operations of the proxies involved in a connection will be invoked with the `NONE_NOW_UPDATES_ON` obtain mode. When this is set to `false`, event type callbacks will be disabled by using the `NONE_NOW_UPDATES_OFF` obtain mode. The default value is `true`.

12.2 Administration Properties

The OpenFusion implementation of the Event Domain Service supports a single administration property, as described in *Table 16*.

Table 16 Event Domain Service Administration Setting

Property	Description
`DomainName`	The name of a domain. This name must be unique within the domain collection of a single factory. A domain name may be useful in some applications as an alternative to an integer domain identifier.

12.3 Exceptions

The exceptions raised by the Event Domain Service are described in *Table 17*.

Table 17 Event Domain Service Exceptions

Exception	Description
`AlreadyExists`	Raised when trying to add a connection that already exists in the target domain. Also raised when trying to create a connection where the source and target channel are the same.
`ChannelNotFound`	Raised when specifying a channel identifier that does not correspond to a channel contained in the target domain. Also raised when trying to get a typed channel using the `get_channel` operation on an untyped event domain.
`ConnectionNotFound`	Raised when trying to get or remove a connection that does not exist in the target domain. Also raised when trying to get a typed connection using the `get_connection` operation on an untyped event domain.
`CycleCreationForbidden`	Raised when the specified connection would form a cycle in the target domain. This exception can only be raised when the `CycleDetection` QoS has been set to `ForbidCycles`.

Table 17 Event Domain Service Exceptions (Continued)

Exception	Description
DiamondCreationForbidden	Raised when the specified connection would form a diamond in the target domain. This exception can only be raised when the `DiamondDetection` QoS is set to `ForbidDiamonds`.
DomainNotFound	Raised when specifying a domain identifier that does not correspond to a domain in a factory collection.
InterfaceNotSupported	Raised when a typed connection is formed between two channels and the specified interface could not be supported by either the source or target channel.
NoSuchImplementation	Raised when a typed connection is formed between two channels and neither channel could find an implementation to support the specified interface.
UnsupportedAdmin	Raised when trying to create a new domain and a specified administration property could not be supported by the Event Domain Service.
UnsupportedQoS	Raised when trying to create a new domain and a specified QoS property could not be supported by the Event Domain Service.

Configuration and Management

13 Notification Service Configuration

13.1 Overview

The configuration of Singleton properties specific to the Notification Service is described in this section. These properties appear in the Administration Manager, a graphical user interface (GUI) based administration tool included with the OpenFusion Graphical Tools.

The Administration Manager can be used to set the Singleton properties. These properties can also be set programatically, generally as described in the service description sections.

Also, the configuration settings enable the Quality of Service and administration properties to be customised when needed.

Details for configuring Persistence, Logging, CORBA, Java and System properties for the Notification Service are described in the *System Guide*.

i Some properties which are not implemented in the initial version 4 release of the Notification Service are shown in the Administration Manager, but are *read-only* or *locked*. These properties are not documented in this guide.

Common Properties

Instances of some common properties are used by a number of different OpenFusion CORBA Services interfaces and services. Settings for these property instances appear in the Administration Manager's Object Hierarchy for the service's Singleton node. This small group of properties are included in this section in order to facilitate configuration of the service while using the Administration Manager. These properties include:

- IOR Name Service Entry
- IOR URL
- IOR File Name
- Resolve Name
- IOR Name Service

13.2 NotificationSingleton Configuration

The Notification Singleton exists as a single object within a given instance of the Notification Service providing the core service functionality

Persistence Properties

Enable Write Ahead Log

When the write-ahead log is enabled, information that is normally written to the underlying database is written to a log file instead. When the log file reaches a specific size (defined by the *Write Ahead Log Maximum Size* property), the database is updated and the log file is reused. The location of the log file is defined by the *Write Ahead Log Directory* property.

The write-ahead log may increase performance when persistent events are required, particularly when events are being delivered quickly (when consumers are available and responding quickly).

The write-ahead log is enabled when this property is set `TRUE` (checked).

Property Name	`DB.WAL`
Property Type	`FIXED`
Data Type	`BOOLEAN`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Write Ahead Log Directory

The directory used to contain write-ahead log files. This directory must be local to the host running the service. The default location is:

`<INSTALL>/domains/<domain>/<node>/NotificationService/data`

where `<INSTALL>` is the OpenFusion installation path. See the *System Guide* for details of the `domains` directory structure.

Property Name	`DB.WAL.Dir`
Property Type	`FIXED`
Data Type	`DIRECTORY`
Accessibility	`READ/WRITE`
Mandatory	`YES`

Write Ahead Log Maximum Size

The maximum number of entries that can be stored in the write-ahead log before flushing (writing to the underlying database) takes place.

Property Name	`DB.WAL.MaxSize`
Property Type	`STATIC`
Data Type	`INTEGER`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Database Plugin Class

This property is used when a database plugin is available to OpenFusion to enhance the event persistence mechanism. Leave this field blank when the plugin is not available.

Property Name	`DB.Plugin`
Property Type	`STATIC`
Data Type	`STRING`
Accessibility	`READ/WRITE`
Mandatory	`NO`

CORBA Properties

The *General* properties are useful for setting the start-up parameters of a Notification Service Singleton object. These properties are all static and mainly read -write. All these properties are optional, but can only be set prior to starting the Notification Service Singleton.

IOR Name Service Entry

The Naming Service entry for the Singleton.

Property Name	`Object.Name`
Property Type	`FIXED`
Data Type	`STRING`
Accessibility	`READ/WRITE`
Mandatory	`NO`

IOR URL

The *IOR URL* property specifies the location of an Interoperable Object Reference (IOR) for the Service, using the Universal Resource Locator (URL) format. This information is used when a client attempts to resolve a reference to the Service. Some examples are:

```
file:/usr/users/openfusion/servers/NotificationService.ior
http://www.prismtech.com/of/servers/NotificationService.ior
corbaloc::server.prismtechnologies.com/NotificationService
```

OpenFusion supports URLs in *Corbaloc*, *Corbaname*, *file*, *FTP* and *HTTP* URL formats, although some ORBs do not support all of these mechanisms. Consult your ORB documentation for specific details.

Property Name	IOR.URL
Property Type	FIXED
Data Type	URL
Accessibility	READ/WRITE
Mandatory	NO

IOR File Name

The *IOR File Name* option specifies the name and location of the IOR file for the Singleton. If this property is not set, the IOR file name will be:

```
<INSTALL>/domains/<domain>/<node>/<service>/<singleton>/<singleton>.ior
```

where <INSTALL> is the OpenFusion installation path. See the *System Guide* for details of the domains directory structure.

Property Name	IOR.File
Property Type	FIXED
Data Type	FILE
Accessibility	READ/WRITE
Mandatory	NO

IOR Name Service

The name of the Naming Service which will be used to resolve the Singleton object.

Property Name	`IOR.Server`
Property Type	`FIXED`
Data Type	`STRING`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Resolve Name

The ORB Service resolution name used to resolve calls to the Singleton.

Property Name	`ResolveName`
Property Type	`FIXED`
Data Type	`STRING`
Accessibility	`READ/WRITE`
Mandatory	`YES`

Messaging Loggers

Service Log File Location

The location of the service log file. Each individual component logger (the scheduler logger, the transaction manager logger, and so on) writes to the same service log file. By default, this is the same log file used at the Service level.

The default location of the service log file is:

```
<INSTALL>/domains/OpenFusion/localhost/NotificationService/
log/NotificationService.log
```

Property Name	`logkit/targets/file/filename`
Property Type	`FIXED`
Data Type	`FILE`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Service Log File Format

The format for entries in the service log file. The default format is:

```
%{priority} [%{category}] %{time:yyyy-MM-dd' 'HH:mm:ss.SSS}
%{message}\n%{throwable}
```

The same format is used by each component logger. This format overrides the format specified in the *Log Pattern* property at the Service level.

Property Name	logkit/targets/file/format
Property Type	FIXED
Data Type	STRING
Accessibility	READ/WRITE
Mandatory	NO

Set All Loggers To

Each component of the Notification Service (the scheduler, the transaction manager, and so on) has its own individual logger. For convenience, every component logger can be set to the same level using this property. Options are:

- *Set all to Disable*
- *Set all to Error*
- *Set all to Warning*
- *Set all to Information*
- *Set all to Debug*
- *Set Individually*

The default level is *Set Individually.*

For fine-grained control over logging, set this property to *Set Individually.* This allows each individual logger to be configured using the individual properties on this tab (described below).

Property Name	GlobalSetting
Property Type	FIXED
Data Type	ENUM
Accessibility	READ/WRITE
Mandatory	NO

Scheduler Logger Level

The logger level for the scheduler. Options are:

- *Disable (0)*

- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/scheduler`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Role Manager Logger Level

The logger level for the role manager. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/rolemanager`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

JTO Logger Level

The logger level for JTO. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*

- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/jto`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Messenger Logger Level

The logger level for the messenger. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/messenger`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Task Manager Logger Level

The logger level for the task manager. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/taskmanager`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

ORB Logger Level

The logger level for the ORB. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/orb`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Transaction Manager Logger Level

The logger level for the transaction manager. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/transactionmanager`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Blobstore Logger Level

The logger level for the blobstore. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/blobstore`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

State Factory Logger Level

The logger level for the state factory. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/statefactory`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

State Machine Factory Logger Level

The logger level for the state machine factory. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/statemachinefactory`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Thread Pool Logger Level

The logger level for the thread pool. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/threadpool`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Notification Service Logger Level

The logger level for the event channel factory (which is the root object of the Notification Service). Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/ecfc`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Component Manager Logger Level

The logger level for the component manager. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/ecm`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Lock Set Factory Logger Level

The logger level for the lock set factory. Options are:

- *Disable (0)*
- *Error (1)*
- *Warning (2)*
- *Information (3)*
- *Debug (4)*

The default level is *Warning*.

Property Name	`logcategory/locksetfactory`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Instrumentation Properties

The interfaces for setting the instrumentation properties, as well as the datatypes for values returned by the `Process.getValue()` method of the CORBA `Process` interface, are given below.

For information on managing instrumentation, including how to obtain the associated property values using the `Process.getValue()` method, please refer to the *System Guide.*

Events Received

This property monitors the total number of all push events received by the Notification Service during execution of the service. In other words, the count of events sent by push suppliers via proxy push consumers.

Property Name	`EventsReceived`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Number of Proxy Push Consumers

This property monitors the current number of structured proxy push consumers in existence on the service.

Property Name	`ProxyPushConsumers`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Number of Structured Proxy Push Consumers

This property monitors the current number of structured proxy push consumers in existence on the service.

Property Name	`StructuredProxyPushConsumers`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Number of Sequence Proxy Push Consumers

This property monitors the current number of sequence proxy push consumers in existence on the service.

Property Name	`SequenceProxyPushConsumers`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Events Delivered

This property monitors the total number of all push events delivered by the Notification Service during execution of the service. In other words, the count of events received by push consumers via proxy push suppliers.

Property Name	`EventsDelivered`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Number of Consumer Admins

This property monitors the current number of consumer admins in existence on the service.

Property Name	`ConsumerAdmins`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Current Total of Events in Channels

This property monitors the total number of events in channels.

Property Name	`CurrentEvents`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Current Total of Events Awaiting Delivery

This property monitors the total number of events awaiting delivery. This count gives the current load on the Service.

This figure is calculated as follows:

Events in queues + (*Events in channel* * *Number of proxies*)

Where:

- *Events in queues* is the number of events in the queues of all proxy suppliers (events which the proxy suppliers have yet to send to their consumer clients).
- *Events in channel* is the number of events in the channel (events which are waiting to be sent to proxy suppliers). This is the count returned by the *Current Total of Events in Channel* property.
- *Number of Proxies* is the number of proxy suppliers.

Property Name	`EventsAwaitingDelivery`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Number of Proxy Push Suppliers

This property monitors the current number of proxy push supplier objects in existence on the service.

Property Name	`ProxyPushSuppliers`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Number of Structured Proxy Push Suppliers

This property monitors the current number of structured proxy push supplier objects in existence on the service.

Property Name	`StructuredProxyPushSuppliers`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Number of Sequence Proxy Push Suppliers

This property monitors the current number of sequence proxy push supplier objects in existence on the service.

Property Name	`SequenceProxyPushSuppliers`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Reconnecting Consumers

This property monitors the current number of unavailable push consumer objects in existence on the service.

Property Name	`ReconnectingConsumers`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Number of Supplier Admins

This property monitors the current number of Supplier Admin objects in existence on the service.

Property Name	`SupplierAdmins`
Property Type	`DYNAMC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

Number of Event Channels

This property monitors the current number of Event Channel objects in existence on the service.

Property Name	`Channels`
Property Type	`DYNAMIC`
Data Type	`COUNTER`
Accessibility	`READ ONLY`
Mandatory	`NO`
`getValue()` Return Type	`longlong`

General Properties

Maximum Queue Size

The maximum queue size of the event delivery manager. When the maximum queue size is exceeded, events are removed from the queue, oldest first, if the *EventReliability* QoS is set to `BestEffort`. In the case of `Persistent`, the events are stored and re-sent when appropriate.

Property Name	`MaxQueueSize`
Property Type	`STATIC`
Data Type	`INTEGER`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Messaging

JMX Instrumentation: Start SUN HTML Adapter

Checkbox. If this is true (checked), then the Sun HTML Adapter will be started alongside the service. The Adapter runs for as long as the notification service does.

The Sun HTML Adapter is a utility provided by Sun that allows JMX instrumentation values to be examined via a web browser. It is provided as an alternative to the *Instrumentation* panel for the Notification Singleton. To use the adapter, specify the port on which it will be run (*JMX Instrumentation: Port for Sun HTML Adapter*) and ensure it is started with the service (*JMX Instrumentation: Start SUN HTML Adapter*). The adapter can be accessed by entering *http://server:port* in a web browser, where

- *server* is the server on which the notification service is running and
- *port* is the port selected for the adapter.

JMX Instrumentation: Port for Sun HTML Adapter

A numeric value which specifies which port the Sun HTML Adapter will run on.

JMX Instrumentation: Register Individual Objects

This is a checkbox: if set then the JMX instrumentation will be available on individual objects (channels, admins and proxies). The *Instrumentation* panel for the Notification Singleton will always display the total figures for the entire

Notification Service. However, these figures are derived from the objects within the service: this control allows those objects to be registered individually when examining using the Sun HTML Adapter, for example.

Lock Set Factory: Fairness Policy

The fairness policy for the lock set factory. Options are:

- *FIFO*
- *JVM*

Although *JVM* is shown as an option, it is not implemented in the initial version 4 release. *FIFO* will be be used, regardless of which option is selected for this property.

Property Name	`components/LockSetFactory/fairness`
Property Type	`FIXED`
Data Type	`ENUM`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Task Manager: Period

The frequency with which the task manager reschedules tasks, expressed in milliseconds. The default is `1000` milliseconds (1 second).

Property Name	`components/TaskManager/period`
Property Type	`FIXED`
Data Type	`INTEGER`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Task Manager: Maximum Activity

This property specifies the number of tasks that the task manager will attempt to keep running at any point in time. This acts as a “soft” limit on thread resource usage. The default number is `10`.

Property Name	`components/TaskManager/maxactivity`
Property Type	`FIXED`

Data Type	INTEGER
Accessibility	READ/WRITE
Mandatory	NO

Task Manager: Priority

The priority for the task manager's own thread. This must be in the range `1 - 10`. The default value is `6`.

Property Name	components/TaskManager/priority
Property Type	FIXED
Data Type	INTEGER
Accessibility	READ/WRITE
Mandatory	NO

Thread Pool: Minimum Pool Size

The minimum pool size for the thread pool. The default is `0` (zero).

Property Name	components/ThreadPool/pool-min
Property Type	FIXED
Data Type	INTEGER
Accessibility	READ/WRITE
Mandatory	NO

Thread Pool: Maximum Pool Size

The maximum size of the thread pool. The default is `20`.

Property Name	components/ThreadPool/pool-max
Property Type	FIXED
Data Type	INTEGER
Accessibility	READ/WRITE
Mandatory	NO

Thread Pool: Initial Pool Size

The initial size for the thread pool. The default is `0` (zero).

Property Name	`components/ThreadPool/pool-initial`
Property Type	`FIXED`
Data Type	`INTEGER`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Thread Pool: Thread Timeout

How long, in milliseconds, an idle thread remains in the pool before being discarded. This controls how long an The default timeout is `1000` milliseconds (1 second).

Property Name	`components/ThreadPool/thread-timeout`
Property Type	`FIXED`
Data Type	`INTEGER`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Transaction Manager: Domain Timeout

The maximum time allowed before a transaction times out, in milliseconds. The default timeout is `60000` milliseconds (60 seconds).

Property Name	`components/TransactionManager/domain/timeout`
Property Type	`FIXED`
Data Type	`INTEGER`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Event Database: Purge Rate

The threshold for the number of *Delete Event* records that can be written to the database before a purge attempt will be initiated. The default value is `1000`.

The purge involves a scan of the database to determine if records are eligible for deleting. An event will be deleted if it has been received and acknowledged by all the consumers who were expected to receive it or if it was discarded by the service.

Property Name	`components/EventDatabase/purgerate`
Property Type	`STATIC`
Data Type	`INTEGER`
Accessibility	`READ/WRITE`
Mandatory	`YES`

Event Database: Maximum Purge Memory

The maximum amount of memory the purge algorithm is allowed to use for storing records in memory during processing, expressed in Kb. The default value is `5000`.

The purge algorithm attempts to match *Store* records with *Delete* records for a specific event and will continue to read records until a match is made or the size of the temporary collection in memory reaches the size set by this property. When this memory threshold is reached, all the records currently in memory are processed and any outstanding records are written to the end of the data files for future processing.

Property Name	`components/EventDatabase/maxpurgememory`
Property Type	`STATIC`
Data Type	`INTEGER`
Accessibility	`READ/WRITE`
Mandatory	`YES`

Journal: Guaranteed Syncing

If set to `true`, this property forces the *Journal* to synchronize the disk file with the Journal file stream when event records are written. If `false`, there is no guarantee that event records will be written to disk (the synchronization will be determined by the JVM).

The default value of this property is `false`.

Property Name	`components/Journal/guaranteedsyncing`
Property Type	`STATIC`

Data Type	BOOLEAN
Accessibility	READ/WRITE
Mandatory	YES

13.3 ProcessSingleton Configuration

IOR Name Service Entry

The Naming Service entry for the Singleton.

Property Name	Object.Name
Property Type	FIXED
Data Type	STRING
Accessibility	READ/WRITE
Mandatory	NO

IOR URL

The *IOR URL* property specifies the location of an Interoperable Object Reference (IOR) for the Service, using the Universal Resource Locator (URL) format. This information is used when a client attempts to resolve a reference to the Service. Currently only *http* and *file* URLs are supported, for example:

```
file:/usr/users/openfusion/ProcessSingleton.ior
http://www.prismtechnologies.com/openfusion/ProcessSingleton.ior
```

Property Name	IOR.URL
Property Type	FIXED
Data Type	URL
Accessibility	READ/WRITE
Mandatory	NO

IOR File Name

The *IOR File Name* option specifies the name and location of the IOR file for the Singleton. If this property is not set, the IOR file name will be:

```
<INSTALL>/domains/<domain>/<node>/<service>/<singleton>/<singleton>.ior
```

where <INSTALL> is the OpenFusion installation path. See the *System Guide* for details of the domains directory structure.

Property Name	IOR.File
Property Type	FIXED
Data Type	FILE
Accessibility	READ/WRITE
Mandatory	NO

IOR Name Service

The name of the Naming Service which will be used to resolve the Singleton object.

Property Name	IOR.Server
Property Type	FIXED
Data Type	STRING
Accessibility	READ/WRITE
Mandatory	NO

14 Notification Service Manager

14.1 Overview

The Notification Service browser acts as a window on to the functioning processes of the service. The Notification Service Manager enables developers to create Event Channels, Admin Objects, and Proxy Objects. A useful feature of the Notification Service Manager is its use in verifying new Notification-Service-based clients.

The Notification Singleton object acts as the base process for a single instance of the OpenFusion Notification Service. The Notification Service Manager is invoked by right-clicking on the Notification Singleton of a running Notification Service in the Administration Manager.

14.2 Using the Notification Service Manager

Start the Notification Service Manager from the command line with the following command:

```
% run com.prismt.cos.treebrowser.notification.
  NotificationServiceBrowser -name NotificationService
```

The Structured Consumer Manager can be started with the following command:

```
% run com.prismt.cos.CosNotification.util.Consumer
  -name NotificationService
```

The Structured Supplier Manager can be started with the following command:

```
% run com.prismt.cos.CosNotification.util.Supplier
  -name NotificationService
```

The Notification Service must be running before any of the Managers can be started.

The Notification Service Manager

The Notification Service Manager displays information about the channels that have been created by an `EventChannelFactory` object. When the manager is first run, and providing no developers have created Event Channels programmatically, the manager will display the default service `EventChannelFactory` object, below the Notification Service icon itself (*Figure 20*).

If the *ChannelConfigurator* Object is present, a saved configuration may be loaded.

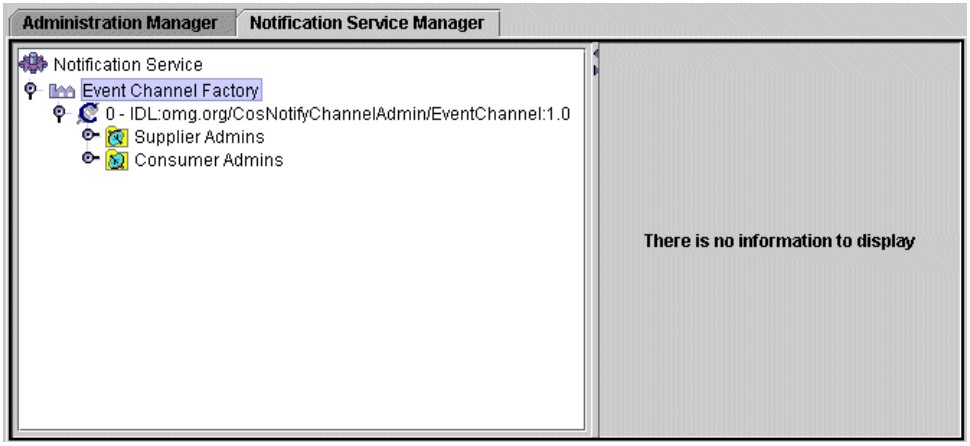

Figure 20 Notification Service Manager

Notification Service Hierarchy

The left-hand pane of the Notification Service browser displays the Notification Service object hierarchy. The icons used in the Notification Service object hierarchy are shown in *Table 18*.

Table 18 Notification Service Nodes

Icon	Object
	*Event Channel Factory* The root node. Also used to show the Default Filter Factory parent node and for Filter Factory objects.
	*Channel* Shows the unique identification number and the name of the channel interface.

Table 18 Notification Service Nodes (Continued)

Icon	Object
	*Supplier Admins* Parent node for all supplier admins.
	*Consumer Admins* Parent Node for all consumer admins.
	*Supplier Admin* Shows the unique identification number and the name of the supplier admin interface.
	*Consumer Admin* Shows the unique identification number and the name of the consumer admin interface.
	*Filters* Parent node for event filters.
	*Proxy Push Suppliers* Parent Node for Proxy Suppliers.
	*Proxy Push Consumers* Parent node for Proxy Consumers.
	*Proxy Push Supplier* Shows the unique identification number and the name of the proxy interface.
	*Proxy Push Consumer* Shows the unique identification number and the name of the proxy interface.

Notification Service Details

The right hand pane will display the details of the individual objects in the hierarchy when they are selected. If no node is selected, or if a node which has no associated details is selected, this box will be empty and contain the message *There is no information to display.*

Setting up an Event Channel

The core component of the Notification Service is the Event Channel. The Event Channel handles the transmission of events over the distributed network provided by the ORB implementation being used.

Creating an Event Channel

Step 1: To create an Event channel, right-click on the Event Channel Factory node in the hierarchy pane of the browser and select *Create Channel*.

Step 2: A new Event Channel instance will be created. If the Event Channel is selected in the hierarchy pane, the details about its *ID* and *Class* name are displayed at the top, and a tabbed pane with the current *Admin* and *QoS* properties and their values are shown. Details about Event Channel properties are described next.

Setting Properties on an Event Channel

Default properties can be set for an Event Channel. This enables the user to specify how the channel will respond to the events it receives. There are two types of property: Admin properties and QoS properties.

Admin Property Settings

Administrative properties refer to property settings that may be applied *only* to event channel objects. These properties are usually set when an event channel is first created. These settings are typically static in nature although they may be changed during the lifetime of the channel object. The standard administrative properties which can be set through the Notification Service Manager are:

- *MaxQueueLength*
- *MaxConsumers*
- *MaxSuppliers*
- *RejectNewEvents*

See *Administrative Properties* on page 126 for a description of these properties.

QoS Property Settings

The QoS properties which can be set on a event channel through the Notification Service Manager are:

- *ConnectionReliability*
- *EventReliability*

- *MaxEventsPerConsumer*
- *MaxReconnectAttempts*
- *MaximumBatchSize*
- *OrderPolicy*
- *PacingInterval*
- *Priority*
- *ReconnectInterval*
- *Timeout*
- *AutoSequenceBatchSize*
- *AutoSequenceTimeout*
- *PropagateQoS*

See Section 5.1, *Quality of Service Properties*, on page 113 for a description of these properties.

Setting up a Supplier or Consumer Admin

A supplier admin is a representation of a `SupplierAdmin` object created by a particular event channel. A consumer admin is a representation of a `ConsumerAdmin` object created by a particular event channel. Every channel is created with a default `SupplierAdmin` and `ConsumerAdmin` object, which are given IDs of `zero`. To view these, expand the tree in the left pane. You should see a similar structure to that shown in *Figure 21*.

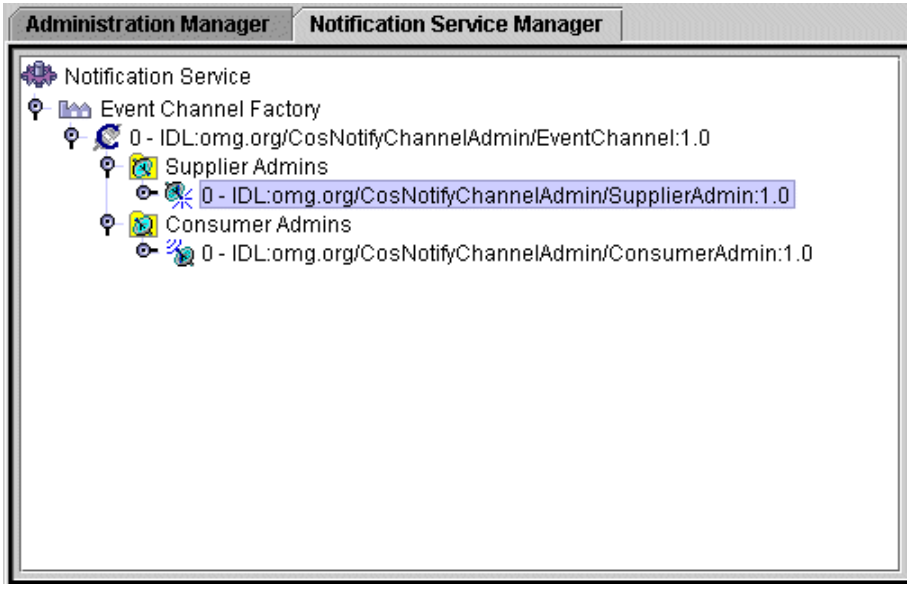

Figure 21 Supplier and Consumer Admins

If the user selects either of the default Supplier or Consumer Admin objects in the hierarchy, then the right panel will display details about these. At the top of the pane there is information about the object selected: its *ID*, *Class*, *Channel* and its default filter operator *OR*. Beneath this is a tabbed panel. One tab displays the *QoS Settings* associated with the object, and the other tab displays *Subscribed Types* (for a *Consumer Admin*) or *Offered Types* (for a *Supplier Admin*).

QoS Settings

The following QoS properties can be set for *SupplierAdmin* and *ConsumerAdmin* objects:

- *ConnectionReliability* (Consumer Admin only)
- *MaxEventsPerConsumer* (Consumer Admin only)
- *MaxReconnectAttempts* (Consumer Admin only)
- *MaximumBatchSize* (Consumer Admin only)
- *OrderPolicy* (Consumer Admin only)
- *PacingInterval* (Consumer Admin only)
- *Priority*
- *ReconnectInterval* (Consumer Admin only)

- *Timeout*
- *AutoSequenceBatchSize*
- *AutoSequenceTimeout*

See Section 5.1, *Quality of Service Properties*, on page 113 for a description of these properties.

Admin Filters

Administration objects and all of the proxy objects in the Notification Service inherit the `FilterAdmin` interface. This means that all of these objects can have filters attached. Each object which can have filters attached contains a child node, *Filters*. The *Filters* node contains children that represent the individual filters that have been created for that object.

Filter Settings

One use of filters is to narrow the sorts of events received by Consumer objects. This is done by applying constraints to Supplier and Consumer Admin objects. These constraints can be specified by using the extended Trader Constraint Language (TCL). To locate the Filter section beneath the Supplier and Consumer Admin objects, expand the hierarchies below each. The Notification Browser should look like that in *Figure 22*.

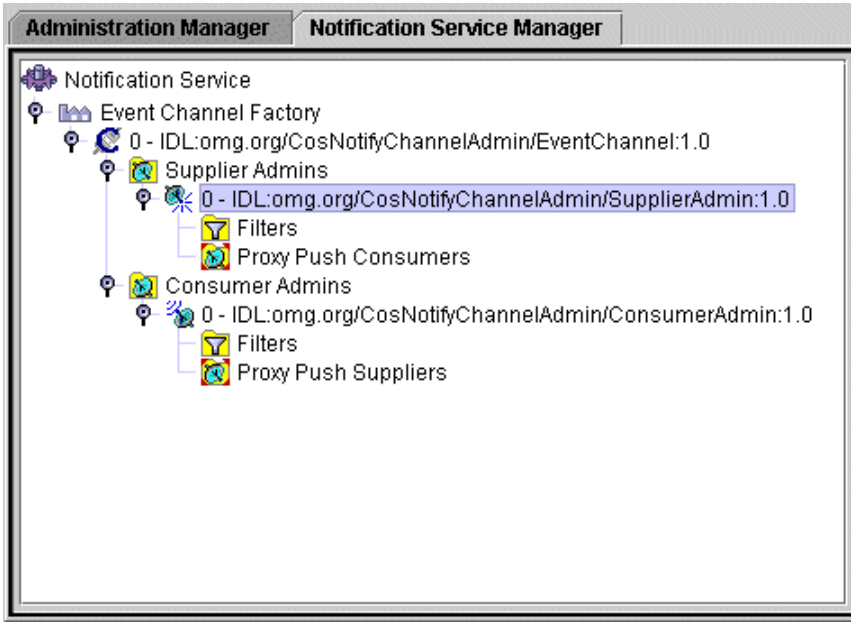

Figure 22 Filters

Custom Filters

A custom filter is a filter which is not based on the standard grammar (TCL) but is created via a custom filter implementation class. This class must implement the `FilterOperations` interface and must be available on the CLASSPATH. The class must be specified when the filter is created, as described in the following section.

Creating a New Filter

Step 1: To create a new filter object, right-click on the *Filters* icon in the hierarchy tree beneath either the Admin or Proxy object. Select the option **Add Filter** from the pop-up menu. The *Add Filter* dialog is displayed, as shown in *Figure 23*.

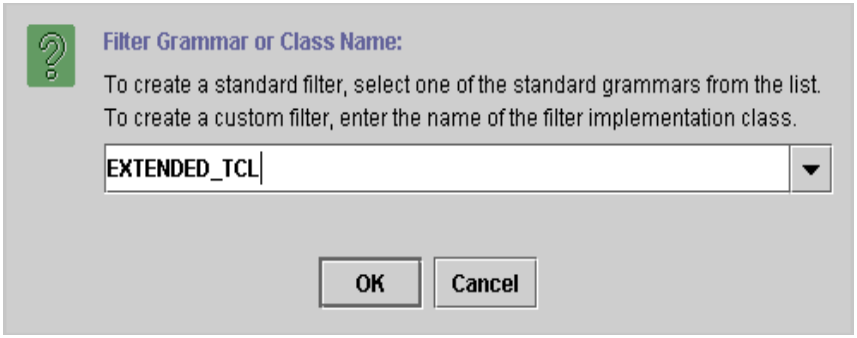

Figure 23 Add Filter

Step 2: Select the required filter grammar from the drop-down list (currently, `EXTENDED_TCL` is the only available option). Or, if a custom filter is required, type the name of the custom filter implementation class into the text box.

Step 3: Click the **OK** button.

Step 4: A new filter object line will appear in the hierarchy. Select this line to view the filter details in the right-hand pane. See *Figure 24*.

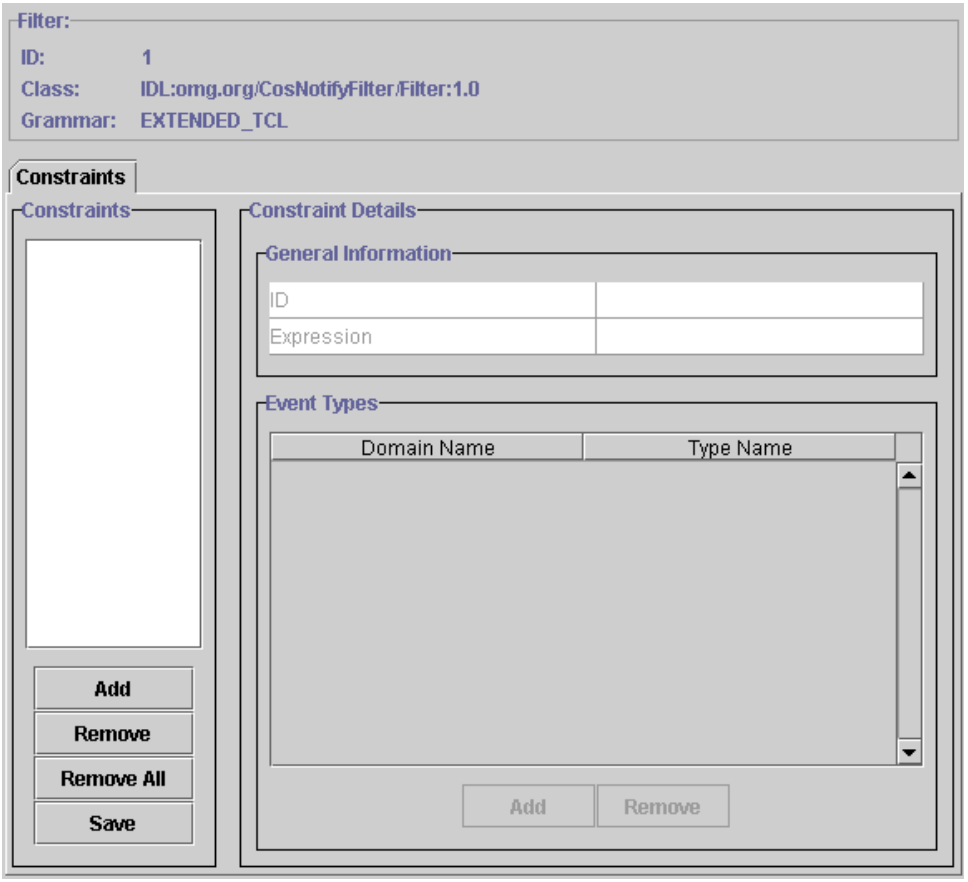

Figure 24 Filter Details

At the top of this filter is a pane containing the filter *ID*, the IDL *Class* on which the filter is based, and the *Grammar* with which it will be constructed. Below this is a split panel. To the left is a pane where any number of filter constraints can be added and removed. To the right is another pane with the details of the constraint currently selected in the left pane.

i If a filter is based on a custom filter implementation class which does not support constraints, the constraint-related controls (Add, Remove) will be disabled.

Adding a Constraint

Step 1: Add a new constraint by clicking the **Add** button in the left pane. This displays the *Add Constraint* dialog, as shown in *Figure 25*.

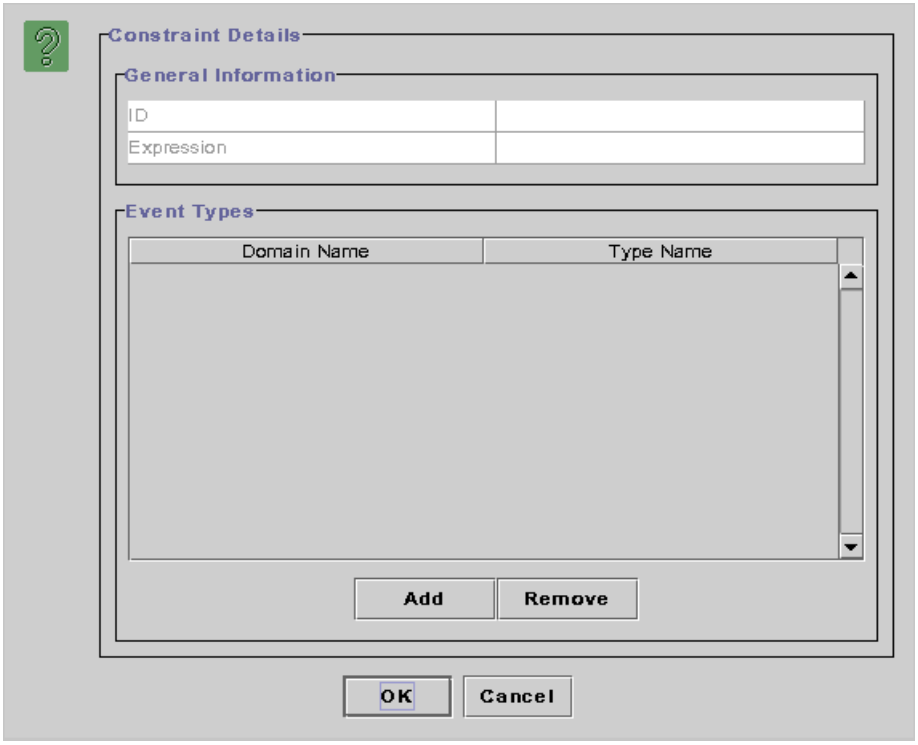

Figure 25 Add Constraint

Each constraint is automatically assigned an ID number. When the constraint is first added, the *ID* text box will be blank.

Constraint expressions are added using the *Expression* field and the *Event Types* table. Steps 2, 3, and 4 illustrate this using the following constraint expression as an example:

```
(($domain_type == 'Telecommunications' and
  $type_name == 'CommunicationsAlarm') or
  $domain_type == 'Healthcare' and
  $type_name == 'VitalSigns')) and severity == 3
```

This expression could be added directly into the *Expression* text box. However it is easier to add the domain and type names of the events into the *Event Types* table.

Step 2: Enter the expression `severity == 3` into the *Expression* text box.

Step 3: Click the **Add** button below the *Event Types* table. A new row will now appear in the table. Enter `Telecommunications` into the *Domain Name* column and `CommunicationsAlarm` into the *Type Name* column.

Step 4: Click the **Add** button below the *Event Types* table and enter `Healthcare` and `VitalSigns` into the *Domain Name* and *Type Name* columns.

Step 5: Click the **OK** button once the full constraint expression has been entered.

Step 6: To complete the process of adding a constraint, click the **Save** button in the *Constraints* panel. The constraint will now be stored.

Removing a Filter

To remove a filter object, right-click on the *Filters* icon in the hierarchy tree beneath the required Supplier or Consumer Admin object. Select **Destroy Filter** from the pop-up menu. A warning dialog will appear to confirm that the filter will now be destroyed and removed from the hierarchy tree.

Removing a Constraint

Step 1: To remove a constraint, select the constraint in the *Constraints* list.

Step 2: Click the **Remove** button below it. The constraint will now disappear from the list. Click the **Remove All** button to remove all constraints from the filter.

Setting Proxy Instances

Supplier and Consumer Proxy objects are shown in the Notification Service Browser beneath Proxy Nodes in the hierarchy panel. See *Figure 26*. A Notification Service may have one or more Proxy instances. These Proxy instances are created using the Supplier or Consumer Admin interfaces.

Proxy instances are used to connect suppliers and consumers to the Event Channel. A supplier connects via a Proxy Consumer, which is obtained from a Supplier Admin. A consumer connects via a Proxy Supplier, which is obtained from a Consumer Admin.

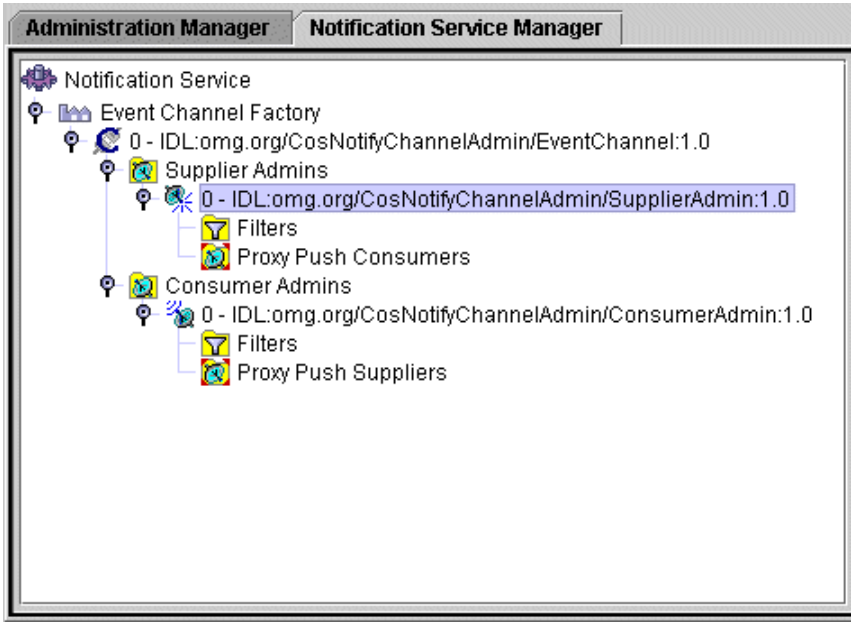

Figure 26 Proxy Objects

QoS Settings

The QoS properties which can be set on a Proxy object through the Notification Service Manager are:

- *ConnectionReliability*
- *DisconnectCallback*
- *MaxEventsPerConsumer*
- *MaxReconnectAttempts*
- *MaximumBatchSize*
- *PacingInterval*
- *Priority*
- *ReconnectInterval*
- *Timeout*
- *AutoSequenceBatchSize*
- *AutoSequenceTimeout*

Some of these Qos properties are not available for all types of Proxy object.

See Section 4.1, *Quality of Service Properties*, on page 69 for a description of these properties.

Creating a New Proxy Object

Supplier Admin objects are used to create proxy consumer objects for Supplier clients. Consumer Admin objects are used to create proxy supplier objects for Consumer clients.

Step 1: To create a new Proxy Object, select the relevant node in the Notification browser hierarchy pane:

- Proxy Push Supplier
- Proxy Push Consumer

Step 2: Right-click on the line in the hierarchy tree and select the **Obtain New Proxy** option from the pop-up menu.

Step 3: Select the Client Type from the list box: *Structured*, or *Sequence*.

Step 4: Click the **OK** button to create the proxy. A new proxy instance will appear in the tree below the node.

Proxy Filters

Proxy objects like Admin objects can have filter objects associated with them. Applying filters to Proxy objects in the Notification Browser is essentially the same process as applying them to Admin objects. Refer to the section *Filter Settings* on page 203 for details.

Upon receipt of each event, the Proxy invokes the appropriate match operation on each of its associated filter objects. The match operation takes the contents of the event as input and returns a boolean result. A FALSE value is returned only when none of the constraints in the filter objects are satisfied by the event, otherwise TRUE is returned. Where the Proxy has multiple filter objects associated with it, it will invoke match on each in turn until either one returns TRUE or all have returned FALSE. Whenever the result of all match operations evaluates FALSE, then the event is discarded.

Testing Event Delivery

The Notification Browser provides facilities for testing the communication between objects in the Notification Service. Once Event Channels are available, the user can configure and create events and send them using built-in Structured Supplier and Consumer clients.

To use the event delivery test clients, the Notification Service requires the following objects to be configured and available.

- An Event Channel object. Refer to *Creating an Event Channel* on page 200.
- Two Event Channel Admin objects. Default Supplier and Consumer Admin objects will always be available when the Event Channel is created, so there is no need to create any more unless the user wishes to do this.

Creating the Test Clients

Once the Notification Service is running and configured correctly, the clients can be created.

- Right click on the *NotificationSingleton* in the Administration Manager's *Object Hierarchy* and select **Notification Structured Supplier Manager** from the pop-up menu. A new *Structured Supplier Manager* will appear as a new tab in the browser framework.
- Right click on the *NotificationSingleton* in the Administration Manager's *Object Hierarchy* and select **Notification Structured Consumer Manager** from the pop-up menu. A new *Structured Consumer Manager* will appear as a new tab in the browser framework.

Configuring the Test Clients

Configuring the Structured Supplier

Figure 27 shows the Structured Supplier Manager. The manager is split into two panes; the *Status* pane and the *Events* pane. The Status pane displays information about the current status of the supplier connection through its proxy and admin objects. The Events pane shows the events being transmitted by the supplier.

The Events pane can be cleared by right clicking on the window and selecting the **Clear** option from the pop-up menu.

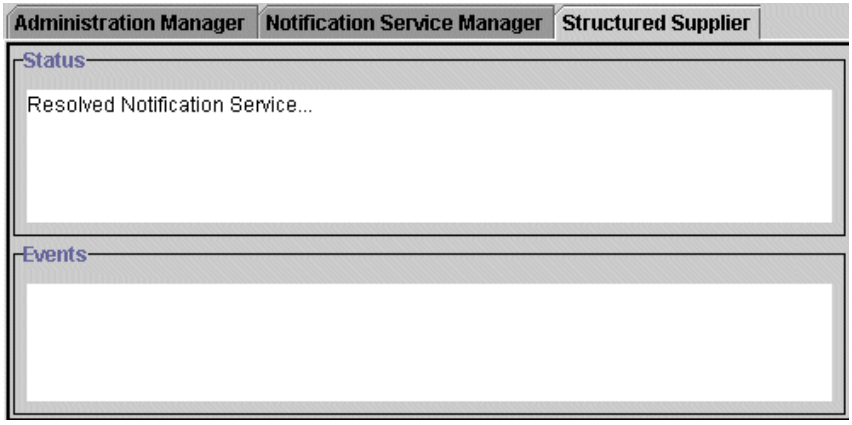

Figure 27 Structured Supplier Manager

Configuring the Structured Consumer

Figure 28 shows the Structured Consumer Manager. The manager is split into two panes; the *Status* pane and the *Events* pane. The Status pane displays information about the current status of the consumer connection through its proxy and admin objects. The Events pane shows the events being received by the consumer.

The Events pane can be cleared by right clicking on the window and selecting the **Clear** option from the pop-up menu.

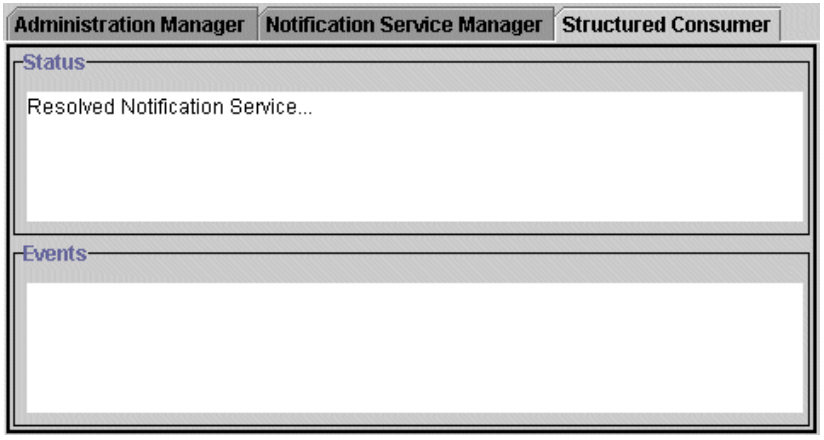

Figure 28 Structured Consumer Manager

Connecting the Structured Supplier

When the Structured Supplier Manager is invoked, the Structured Supplier client resolves the Notification Service.

Step 1: Connect the Structured Supplier to the Notification Service by clicking on the *Connect Supplier* icon in the tool bar. You will then be prompted to select the identifier of the Event Channel and Supplier Admin. If there is more than one Event Channel or more than one Supplier Admin available then you can select the appropriate identifiers from the drop-down lists.

Step 2: Select a Channel and Admin and click **OK**. The Structured Supplier client will now be connected to the Notification Service and will create a proxy automatically.

Connecting the Structured Consumer

When the Structured Consumer Manager is invoked, the Structured Consumer client resolves the Notification Service.

Step 1: Connect the Structured Consumer to the Notification Service by clicking on the *Connect Consumer* icon in the tool bar. You will then be prompted to select the identifier of the Event Channel and Consumer Admin. If there is more than one Event Channel or more than one Consumer Admin available then you can select the appropriate identifiers from the drop-down lists.

Step 2: Select a Channel and Admin and click on OK. The Structured Consumer client will now be connected to the Notification Service and will create a proxy.

Creating Test Events

The final stage of configuration is to create events to transmit over the Notification Service.

Step 1: Click on the Structured Supplier Manager tab in the browser, and click on the **Configure Events** tool bar button. The *Configure Events* dialog box is displayed, as shown in *Figure 29*.

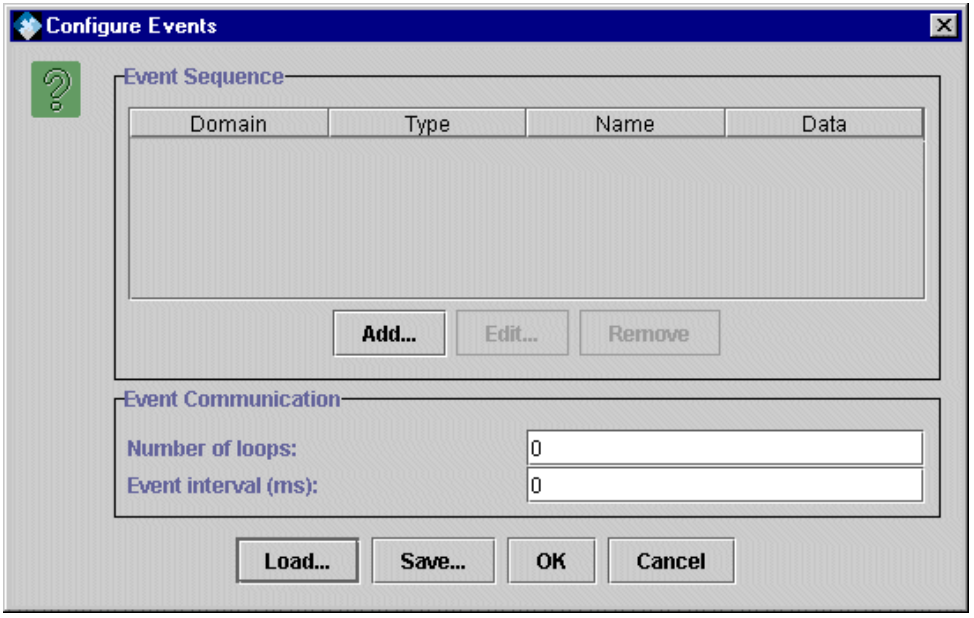

Figure 29 Configure Events Dialog Box

The *Configure Events* dialog is separated into two panes. The *Event Sequence* contains a list of the events to be transmitted. The *Event Communication* allows the user to configure the event transmission mechanism. The *Number of Loops* field expects an integer for the number of times that the batch of events in the *Event Sequence* table will be transmitted across the Event Channel. In normal circumstances events are usually transmitted once only, but for testing purposes this can be increased. *The Event Interval* field allows the user to specify, in milliseconds, the interval between the transmission of the event batches listed in the *Event Sequence* table.

Step 2: Enter the value of `10` into the *Number of Loops* field and `100` into the *Event Interval* field. This will instruct the Notification Service to transmit the event sequence ten times, at intervals of one every one tenth of a second.

Step 3: Click the **Add** button in the *Event Sequence* pane. This gives a dialog box for creating structured events, shown in *Figure 30*.

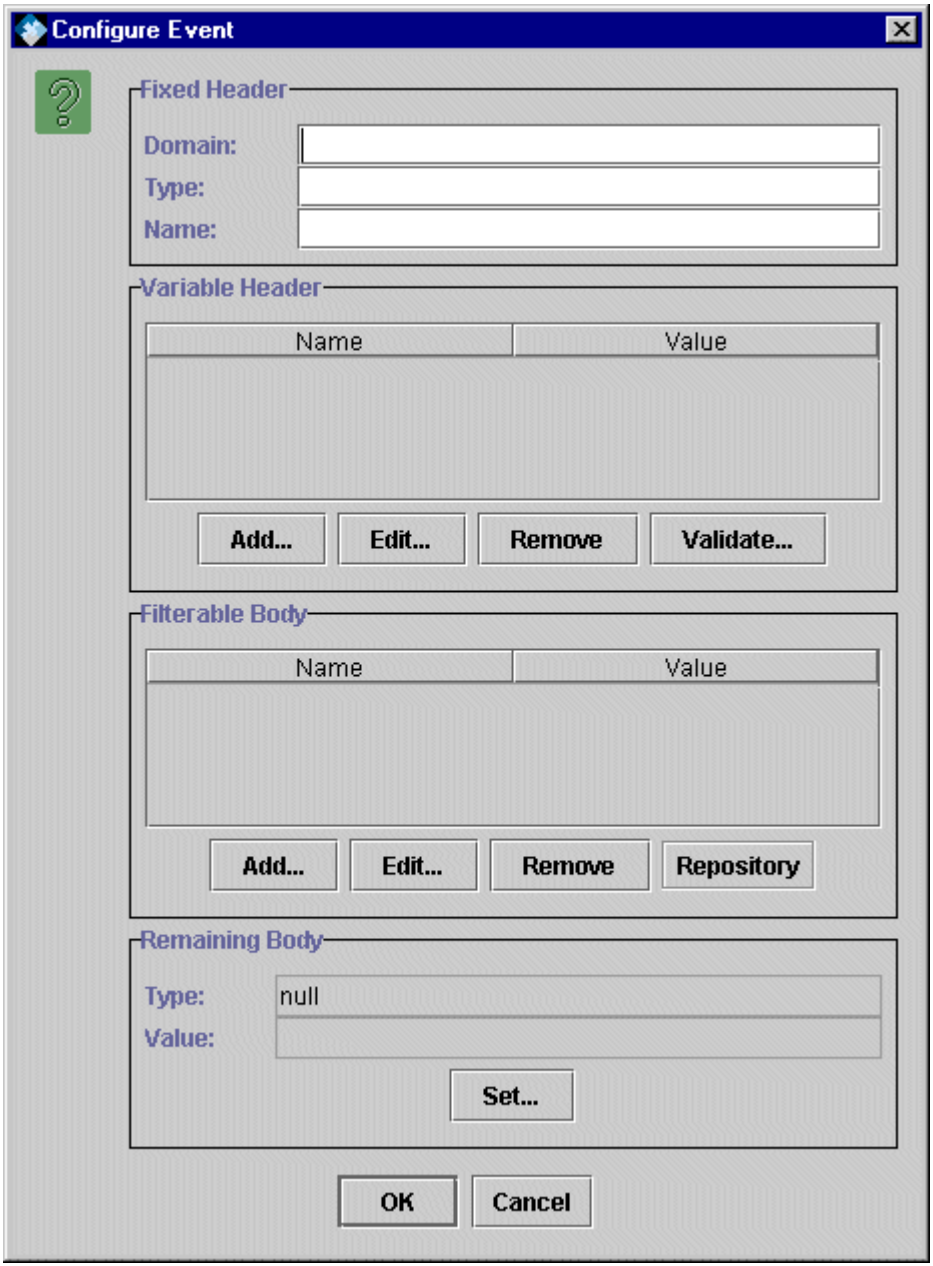

Figure 30 Configure Event Dialog Box

Step 4: Enter `Healthcare` into the *Domain* field of the *Fixed Header* section, and `VitalSigns` into the *Type* field. Enter an identifier for the Event instance (for example, `my_vital_signs_event_1`).

Step 5: Click the **Add** button in the *Filterable Body* section of the dialog. Enter the property `severity` into the *Name* field and switch the data type to `short` in the *Type* field. Finally set the value to `3` in the *Value* field. Click **OK**. The *Filterable Body* will now contain the new property.

Step 6: Click **OK** to load the event into the *Event Sequence* table of the *Configure Events* dialog.

Step 7: Repeat step 3 through step 6 as before, but give this event a different identifier and set the severity to `4`.

Step 8: To save a configured event sequence for use at a later date, click the **Save** button. To load events select the **Load** button and load a previously saved file. For this exercise click on **OK**.

Transmitting Test Events

Step 1: To begin transmitting the events, click the **Send Events** button on the tool bar.

Step 2: If you examine the Structured Supplier Manager you should notice the events being transmitted in the Events pane.

Step 3: If you switch to the tab of the Structured Consumer Manager you will notice the events being received in the Events window.

Filter Events

The next example will demonstrate the use of filters on event transmission.

Step 1: Select the Notification Service Manager window and create a new Filter object on the Supplier Admin object.

Step 2: Create a new constraint.

Step 3: Add the expression `$severity != 3`, and add the domain `Healthcare` and type `VitalSigns` to the *Event Types* table. This will create a filter to *accept only Healthcare/Vital Signs events whose* `severity` *is not equal (*`!=`*) to 3*. Property variables in constraint expressions ***must always*** be preceded by the `$` sign.

Step 4: Clear the *Events* panes in the Structured Supplier and Consumer Manager windows and click the **Send Events** button again.

Step 5: Examine the *Events* pane in the Structured Supplier Manager. Both events are transmitted to the Event Channel.

Step 6: Now examine the *Events* pane in the Structured Consumer Manager. You should notice that only the event with `severity==4` is being received by the Consumer client. The event with `severity==3` is filtered out due to the constraint created on the Supplier Admin in step 3.

Destroying Proxy Objects

Proxy objects are destroyed if the **Disconnect** button is clicked or if the browser is closed.

15 ChannelConfigurator Tool

15.1 Overview

The ChannelConfigurator tool is a Java Object which is used with the Notification Service to help manage channel configurations. The configuration of Notification Service channels can be saved and used to re-initialise the Notification Service when it is restarted. The Service can therefore be stopped and started without the added overhead of recreating all the channels.

The ChannelConfigurator can perform the following functions:

- Save the Notification Service channel configuration into an XML file.
- Load an existing channel configuration into the Notification Service from an XML file.

15.2 ChannelConfiguratorObject Configuration

The ChannelConfigurationObject Java Object must be added to the Notification Service before the ChannelConfigurator tool can be used. Adding Java Objects to a Service is described in the *System Guide*.

Once the ChannelConfigurationObject has been added to the Service, the following properties must be configured before the Notification Service is restarted.

NotificationServiceName

The name of the Notification Service that the ChannelConfigurator tool will run on. The default value is *NotificationService*.

Property Name	`NotificationServiceName`
Property Type	`DYNAMIC`
Data Type	`STRING`
Accessibility	`READ/WRITE`
Mandatory	`YES`

NameServiceName

The name of the Naming Service that the ChannelConfigurator tool will bind objects to.

Property Name	`NameServiceName`
Property Type	`DYNAMIC`
Data Type	`STRING`
Accessibility	`READ/WRITE`
Mandatory	`NO`

Channel Configuration URL

The URL of the XML file containing the channel configuration information. This property is mandatory but does not have a default value, so a value must be entered before the Notification Service can be started.

Property Name	`ChannelConfigurationURL`
Property Type	`DYNAMIC`
Data Type	`STRING`
Accessibility	`READ/WRITE`
Mandatory	`YES`

15.3 Using the ChannelConfigurator Tool

When the Notification Service is started, the ChannelConfigurator tool will automatically attempt to load channel configurations from the XML file pointed to by the *Channel Configuration URL* property. If the file cannot be located, the Service will start with no channels configured.

The tool will attempt to resolve each object described in the XML file, according to the following rules:

1. If the XML file contains an ID number (`ID` element), the tool will load the object described by the ID.
2. If the XML file contains an IOR string (`IOR` element), the tool will load the object described by the string.
3. If the XML file contains an IOR URL (`IOR_URL` element), the tool will load the object pointed to by the URL.
4. If the XML file contains a Naming Service entry (`NS_Entry` element) and the object can be resolved in the Naming Service, the tool will load that object.

5. If the XML file contains a Naming Service entry (`NS_Entry` element) but the object cannot be resolved, the tool will create a new object and register it in the Naming Service with the name specified by the `NS_Entry` element.

These rules are evaluated in the order given. So if all three elements exist for an object, the object will be resolved from the IOR string and the other elements will not be evaluated.

If the tool cannot resolve an object from any of these elements, it will create a new object.

i From version 2.5.3 onwards, only the `ID` element is used. The other elements (`IOR`, `IOR_URL`, and `NS_Entry`) are still checked, but this is only for compatibility with files created by earlier versions (which did not have the `ID` element). It is suggested that older XML files are re-saved in the current version in order to update their structure.

Saving a Channel Configuration

To save the Notification Service's current channel configuration, open the Notification Service Manager. Right-click on the root node of the Notification Service hierarchy and select Save Channel Configuration from the pop-up menu, as shown in *Figure 31*.

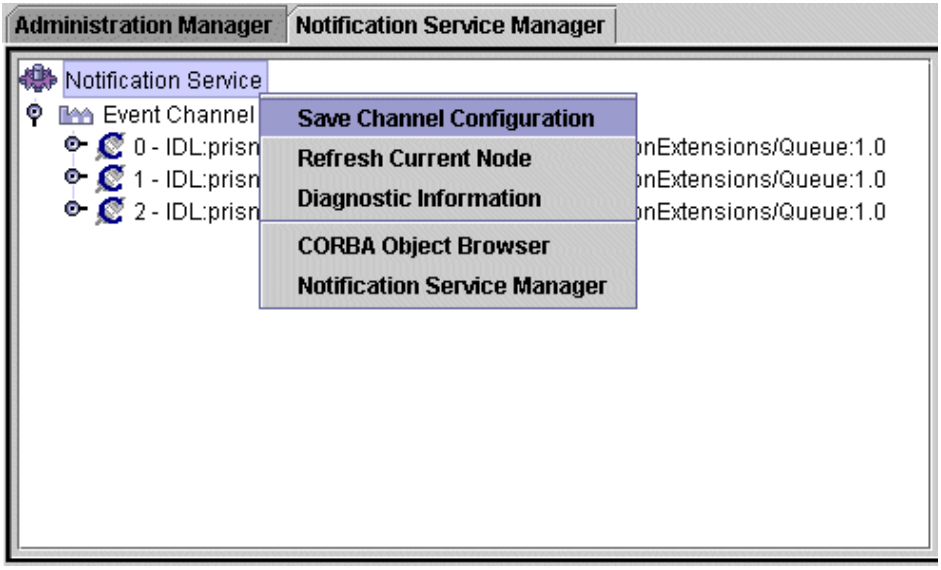

Figure 31 Saving Channel Configuration

A *Save* dialog box is displayed. Select the directory and file name for the XML file. The file should be given an `.XML` extension.

If the specified XML file already exists, it will be overwritten by the new file.

If the file name and location do not match that specified by the *Channel Configuration URL* property, then the Notification Service will *not* be initialised with the saved configuration the next time it is started.

15.4 Running from the Command Line

To load a saved channel configuration into the Notification Service:

```
% run com.prismt.cos.CosNotification.tools.xml.ChannelConfigurator:
  -load <URL> <NotificationService> <NamingService>
```

To save the current channel configuration of the Notification Service to an XML file:

```
% run com.prismt.cos.CosNotification.tools.xml.ChannelConfigurator:
  -save <URL> <NotificationService>
```

Where:

`<URL>` is the URL of the XML configuration file.

`<NotificationService>` is the Notification Service resolve name.

`<NamingService>` is the Naming Service resolve name.

Index

Index

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

W