

opentext™



Discover the Future of CORBA

Micro Focus
OpenFusion CORBA Services
Version 5.1

Notification Service Guide

Copyright 2009-2023 Open Text.

The only warranties for products and services of Open Text and its affiliates and licensors ("Open Text") are as may be set forth in the express warranty statements accompanying such products and services.

Nothing herein should be construed as constituting an additional warranty. Open Text shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.

2023-09-05

Contents

Preface	vii
About the Notification Service Guide	vii
Intended Audience	vii
Organisation	vii
Conventions	vii
Contacting Micro Focus	viii
Further Information and Product Support	viii
Information We Need	ix
Contact information	ix
Introduction	1
Part I Notification Service	
Description	5
OMG Standard Features	5
OpenFusion Enhancements	6
Concepts and Architecture	6
Dependencies on Other Services	7
The Basic Concept	7
The Architecture	8
The Details	11
Structured Events	11
Event Communication Models	12
Event Channel	12
Admin Objects	13
Proxies	13
Queues	14
Quality of Service	15
Filtering	16
Sequencing	17
Persistence	19
Federation	20
Using the Service	23
Introduction	23
Import Statements	23
Compiling and Running Clients	24
Compiling Client Applications	24
Running Client Applications	24
Initialising the ORB	24
Starting the Notification Service	24
Configuring the Notification Service	25
Starting Clients	25
Creating Clients	25
Creating a Supplier	25
Connecting to the Server	25
Creating Events	29
Sending Events	30
Creating a Consumer	31
Connecting to the Server	31
Receiving Events	34
Suspending and Resuming Connections	35

Removing Inactive Proxies.....	35
Proxy Push Consumers	36
Proxy Push Suppliers	36
Alternative Method	36
Using Quality of Service Properties	37
Creating an Event Channel with QoS.....	37
Managing QoS	38
Adding New QoS to a Channel	38
Accessing the QoS	38
Validating Event QoS	39
Using Filters	39
Filter Objects.....	39
Creating a Filter Object.....	40
Adding a Filter Object to an Admin Object	40
Listing Filter Objects.....	40
Removing Filter Objects	41
Event Filters	41
Constructing Constraints	41
Managing Constraints	42
Writing Constraint Expressions	43
Extended TCL Grammar	43
Basic Elements	43
Operators	44
Constraint Examples.....	46
Using Persistence	46

API Definitions 47

OMG Standard API Definitions.....	47
Event Channel Factory Interface.....	49
Event Channel Interface.....	50
Administration Interfaces	51
Filter Interfaces	51

Supplemental Information 53

Quality of Service Properties	53
Standard OMG Properties	53
OpenFusion QoS Extensions.....	58
Memory Management Properties	62
Administrative Properties	64
Errors and Exceptions	65
Errors.....	65
Exceptions	65
Implementation Limit Exception	66

Part II Configuration and Management

Notification Service Configuration 69

Common Properties	69
NotificationSingleton Configuration	69
Persistence Properties.....	69
CORBA Properties.....	70
Messaging Loggers	72
Instrumentation Properties	80
General Properties.....	88
Messaging.....	88

ProcessSingleton Configuration	92
Notification Service Manager	95
Using the Notification Service Manager	95
The Notification Service Manager	95
Notification Service Hierarchy.....	96
Notification Service Details	97
Setting up an Event Channel	97
Creating an Event Channel.....	97
Setting Properties on an Event Channel	97
Admin Property Settings.....	98
QoS Property Settings.....	98
Setting up a Supplier or Consumer Admin	98
QoS Settings.....	99
Admin Filters.....	100
Filter Settings	100
Setting Proxy Instances	103
QoS Settings.....	104
Creating a New Proxy Object.....	104
Proxy Filters	104
Testing Event Delivery	105
Creating the Test Clients.....	105
Configuring the Test Clients	105
Destroying Proxy Objects.....	109
ChannelConfigurator Tool	111
ChannelConfiguratorObject Configuration	111
Using the ChannelConfigurator Tool	112
Saving a Channel Configuration	113
Running from the Command Line	113
Index.....	115

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.

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 OpenFusion 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 (such as Java, IDL) and of the relevant underlying technologies (J2EE, CORBA).

Organisation

The **Notification Service Guide** is organised into two main sections. The first section describes the OpenFusion Notification Service, and 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 second section, "[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. This section should 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**.



i

WIN

UNIX

C

C++

Java

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

Item contains helpful hint or special information.

Information applies to Windows systems only.

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

C language specific

C++ language specific

Java language specific

Hypertext links are shown as [blue](#).

Items shown as cross-references, such as "[Contact information](#)", act as hypertext links; click on the reference to go to the item.

```
% 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 boxes

```
:
NameComponent newName[] = new NameComponent[1];
// set id field to "example" and kind field to an empty string
newName[0] = new NameComponent ("example", "");
```

Italics and ***Italic Bold*** indicate new terms or emphasise an item.

Bold indicates user related actions, e.g. **File | Save** from a menu.

Steps in a task are numbered:

1 One of several steps required to complete a task.

Contacting Micro Focus

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

Further Information and Product Support

Additional technical information or advice is available from several sources.

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

- The *Product Updates* section of the Micro Focus SupportLine Web site, where you can download fixes and documentation updates.
- The *Examples and Utilities* section of the Micro Focus SupportLine Web site, including demos and additional product documentation.

To connect, enter <http://www.microfocus.com> in your browser to go to the Micro Focus home page, then click *Support*.

Note:

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

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

Also, visit:

- The Micro Focus Community Web site, where you can browse the Knowledge Base, read articles and blogs, find demonstration programs and examples, and discuss this product with other users and Micro Focus specialists.
- The Micro Focus YouTube channel for videos related to your product.

Information We Need

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

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. You can find this by either logging into your order via the Electronic Product Distribution email or via the invoice with the order.

Contact information

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

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the *Product Updates* section of the Micro Focus SupportLine Web site, where you can download fixes and documentation updates. To connect, enter <http://www.microfocus.com> in your browser to go to the Micro Focus home page, then click *Support*.

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

You may want to check in particular:

- <https://supportline.microfocus.com/productdoc.aspx>. (documentation updates and PDFs)

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

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

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.

Part I

Notification Service

In this part

This part contains the following:

Description	page 5
Using the Service	page 23
API Definitions	page 47
Supplemental Information	page 53

Description

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 (that is, 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 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 channels* 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
- 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 controllability, 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

Limitations

This version of the OpenFusion Notification Service does not support the OMG-defined *pull* model since the pull model is rarely used. Removing this model has enabled the OpenFusion Notification Service to be smaller and have better performance.

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 (such as a new alarm) is added, then all event user, or *consumer*, objects (such as 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 (the *consumers*) will need to be altered, re-compiled, tested, etc., whenever supplier objects (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, that is, 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 Implementation”, shows that only the *push* model of event transmission is used in the OpenFusion implementation of the basic architecture.

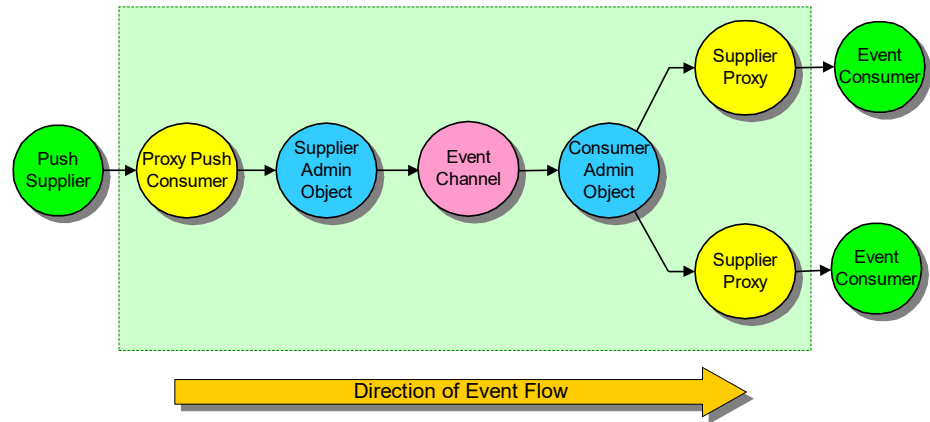


Figure 1 Basic OpenFusion Implementation

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, and queues

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 used by the OpenFusion Notification Service is the push model.

i

Note:

The OMG-defined *pull* model is rarely used and was removed from the OpenFusion Notification Service in order to reduce size and complexity as well as improving performance. This version of the OpenFusion Notification Service does not support the pull model.

Figure 2, “Main Components”, shows the service’s main components, including filters, queues, and translation.

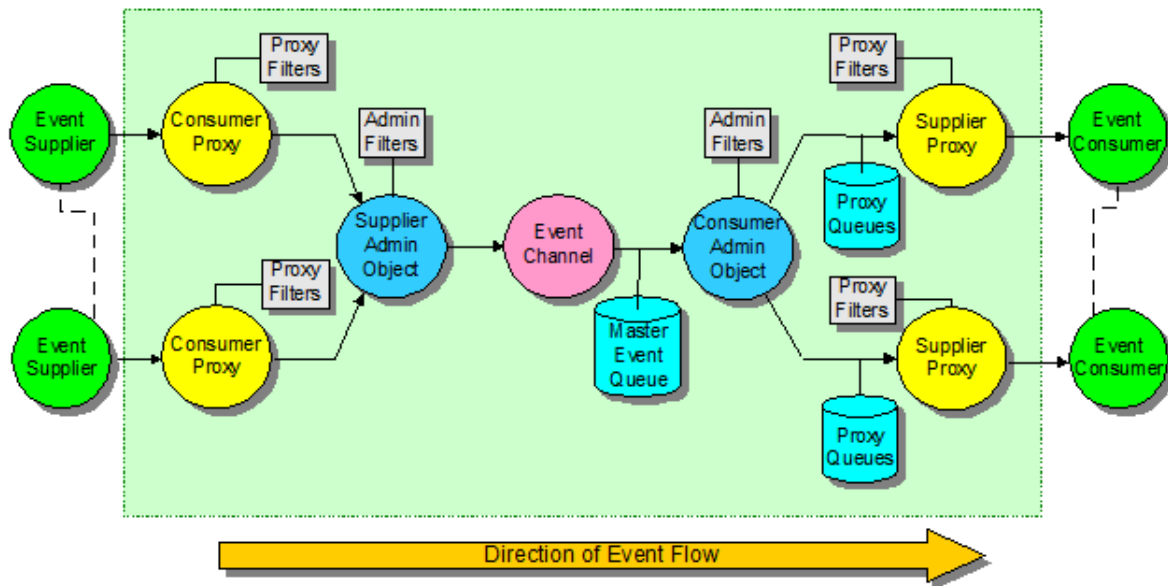


Figure 2 Main Components

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*

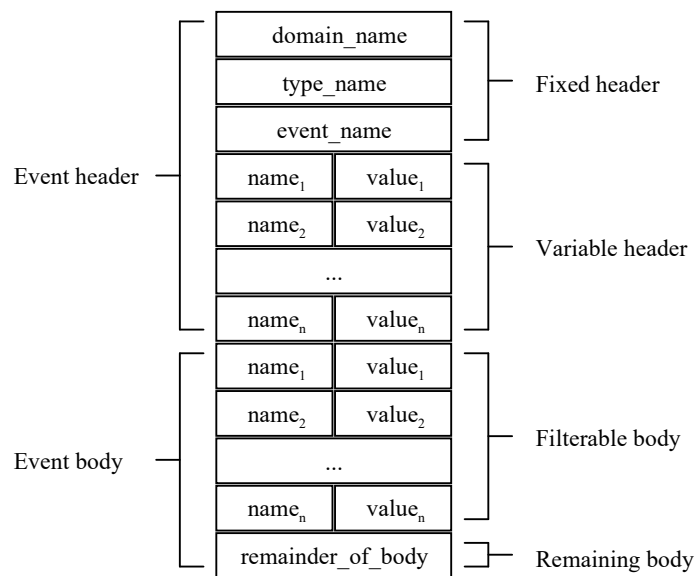


Figure 3 Structured Event

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 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” 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”.)

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 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 (that is, 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, a *consumer proxy* (also called *proxy consumer*: both forms are used in the OMG specification). 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 supplier) 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.

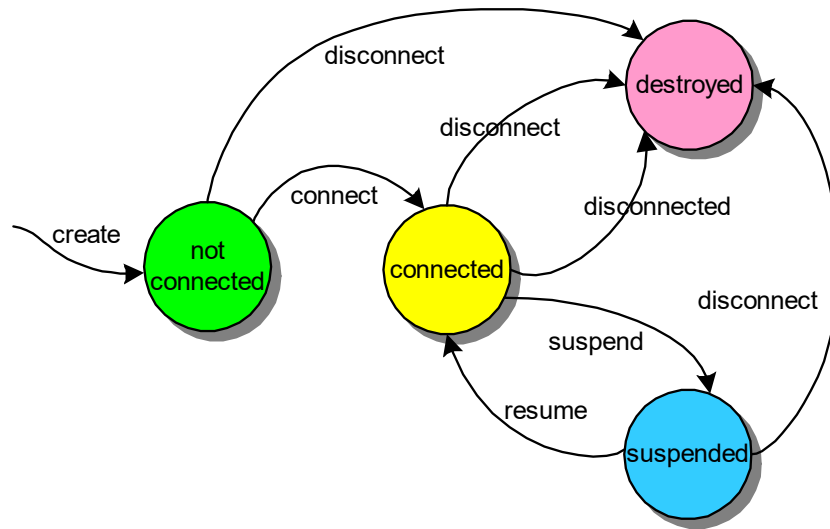


Figure 4 Proxy States

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 (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.

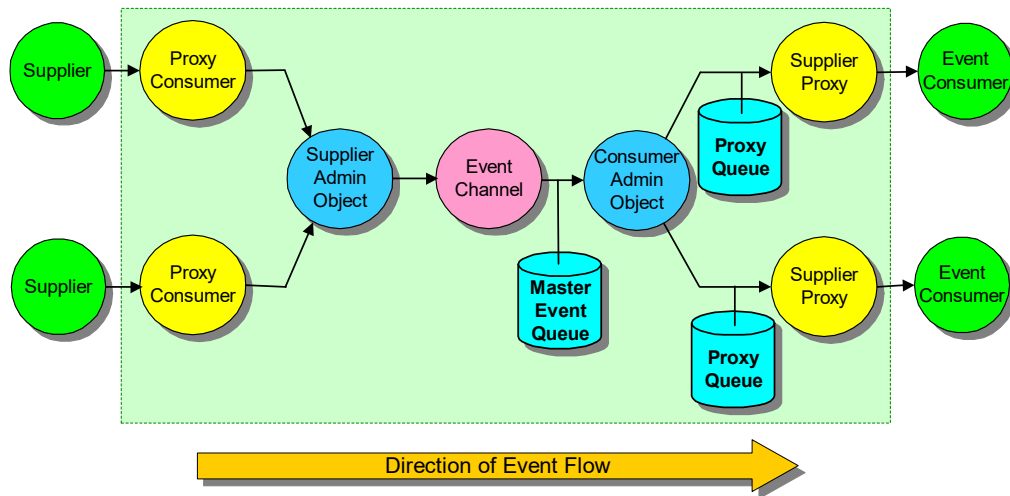


Figure 5 Event Queues

Quality of Service

- 1 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:
 - a When the event is delivered by the supplier to the event channel
 - b When it is forwarded by the channel
 - c 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 in the “[Standard OMG Properties](#)” section of the “[Supplemental Information](#)” chapter.

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 in “[OpenFusion QoS Extensions](#)” in the “[Supplemental Information](#)” chapter.

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”](#)). 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”](#).

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.

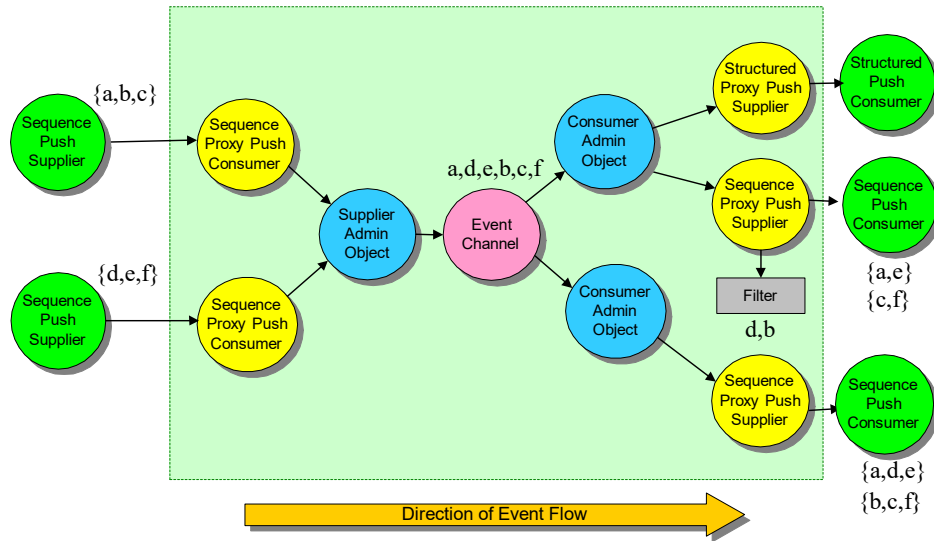


Figure 6 Sequencing Architecture

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 should 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 **off** in an OpenFusion installation. If it is required, it should be switched on using the appropriate QoS settings (as described in “[AutoSequenceBatchSize](#)”).

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.

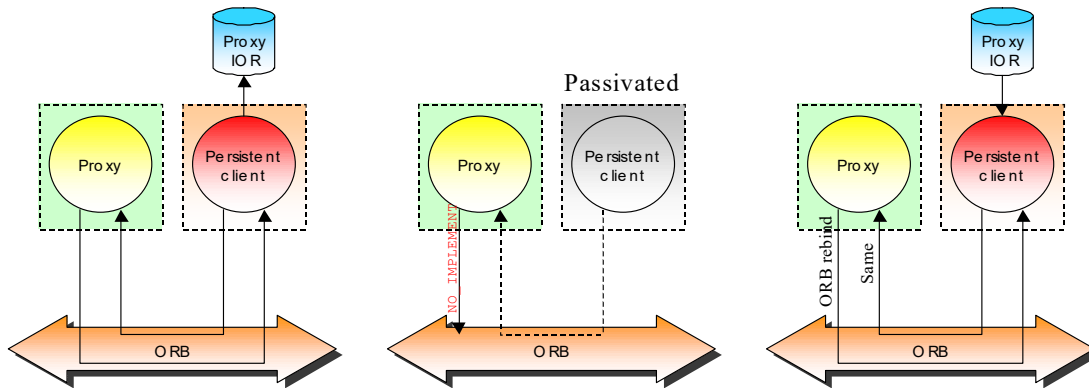


Figure 7 Passivating Persistent Clients

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.

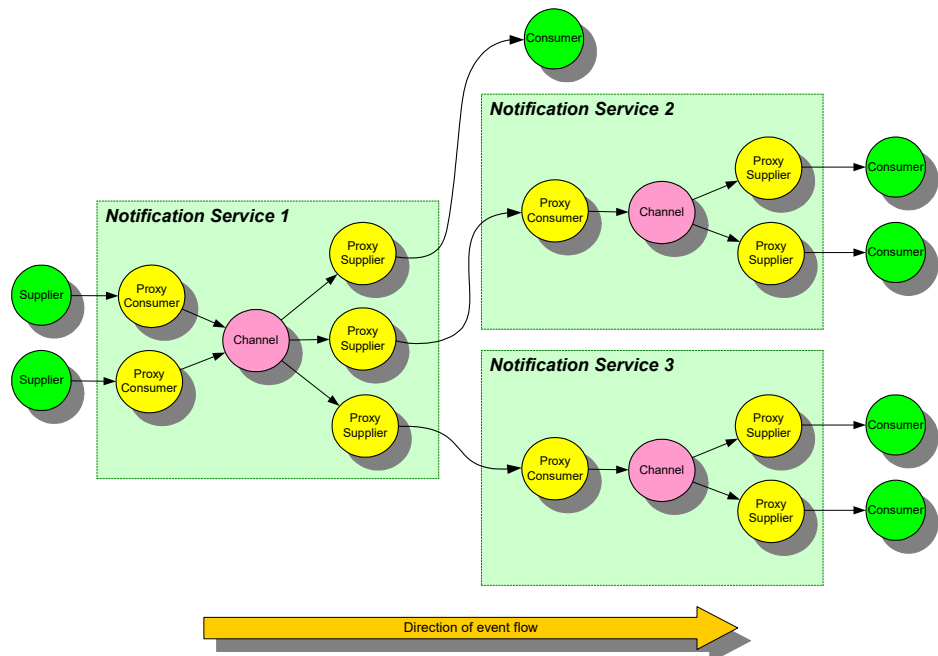


Figure 8 Federation of Channels Architecture

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 intermediary, 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.

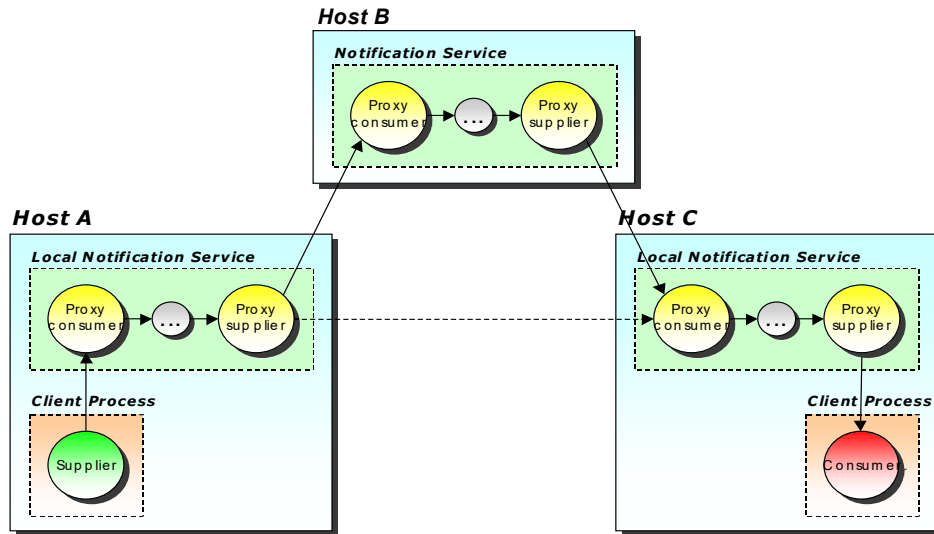


Figure 9 Local Host

Using the Service

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.*
```

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. There 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.

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

Starting the Notification Service

- 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.
- 2 Ensure the appropriate ORB daemon is running (see above).
- 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. The section “[Notification Service Configuration](#)” 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.

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

- 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);

```

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"](#).

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.

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:

- a **AND**: Both an administration filter and a proxy filter must pass an event in order for the event to be forwarded.
- b **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);
}
```

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:

- a Obtains the unique identifier, the channel and the filter operation
- b Lists the total number of proxies
- c Examines whether or not the proxy with identifier 42 exists for a SupplierAdmin object called *admin*

```
int id = admin.MyID ();
EventChannel ec = admin.MyChannel ();
InterFilterGroupOperator op = admin.MyOperator ();
```

```

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 doesnt exist!);
}

```

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;
}

```

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”](#).

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 qos[] = new Property [2];
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

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, qos);
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 29 for an example of how to create a structured event.

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

- 1 Resolve an event channel factory. Code for this is given in "[Connecting to the Server](#)", step 1 on [page 25](#).
- 2 Obtain a reference to an event channel. Code for this is given in "[Connecting to the Server](#)", step 2 on [page 26](#).
- 3 Obtain a reference to a supplier admin object. Code for this is given in "[Connecting to the Server](#)", step 3 on [page 27](#).
- 4 Obtain a reference to a proxy consumer object. Code for this is given in "[Connecting to the Server](#)", step 4 on [page 27](#).
- 5 Connect to the proxy consumer. Code for this operation is given in "[Connecting to the Server](#)", step 5 on [page 29](#).

- 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.remainer_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

- 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”](#):

```
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);
```

- 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”](#):

```
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) {}

```

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:

- a **AND**: Both an administration filter and a proxy filter must pass an event in order for the event to be forwarded.
- b **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);
}

```

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:

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

```
int id = admin.MyID ();  
EventChannel ec = admin.MyChannel ();  
InterFilterGroupOperator op = admin.MyOperator ();  
  
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 doesnt exist!);
}
```

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;
}
```

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”](#).

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 [“Quality of Service Properties”](#) 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

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.

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 JacORB 3.9 ORB throws `OBJECT_NOT_EXIST` correctly.

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 VisiBroker 8.5 ORB has this behaviour.

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 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](#)” 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.

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 either a supplier or a consumer.

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

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

- 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*.

```
qosProp[0] = new Property ();  
qosProp[0].name = EventReliability.value;  
qosProp[0].value = orb.create_any ();  
qosProp[0].value.insert_short (org.omg.CosNotification.Persistent.value);
```

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

```
qosProp[1] = new Property ();  
qosProp[1].name = ConnectionReliability.value;  
qosProp[1].value = orb.create_any ();  
qosProp[1].value.insert_short (org.omg.CosNotification.Persistent.value);
```

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

- 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 (qosProp, admProp, 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"](#).

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 qosP[] = channel.get_qos ();
Property admP[] = channel.get_admin ();

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

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

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.
}
```

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.

- 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 ();
```

- 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**.

- 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 ("*", "*");
```

- 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 43.

- 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);
```

- 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:

<code>==</code>	equality
<code>!=</code>	inequality
<code>></code>	greater than
<code>>=</code>	greater than or equal
<code><</code>	less than
<code><=</code>	less than or equal
<code>~</code>	substring match
<code>in</code>	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"](#) 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 `$.header.fixed_header.event_type.type_name == 'Info'`
- The expression can be simplified using runtime variables ([page 44](#)) to give:
- `$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'`

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***.

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 5 of the OpenFusion Notification Service is the same in as in previous versions, however features which are not supported in version 5 throw a `NO_IMPLEMENT` system exception.

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
<i>AdminPropertiesAdmin</i>	A base interface for the <code>EventChannel</code> interface which supports operations for setting and getting various administrative properties on an event channel object.
<i>QoSAdmin</i>	A base interface for the <code>EventChannel</code> 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
<i>PushConsumer</i>	An interface for untyped push consumers. The Notification Service version of this interface supports the <code>PushConsumer</code> interface from the Event Service as well as the <code>NotifyPublish</code> interface.
<i>PushSupplier</i>	An interface for untyped push suppliers. The Notification Service version of this interface supports the <code>PushSupplier</code> interface from the Event Service as well as the <code>NotifySubscribe</code> interface.
<i>SequencePushConsumer</i>	An interface for sequence style push consumers.

Table 2 CosNotifyComm Interfaces (Continued)

Interface	Purpose
<i>SequencePushSupplier</i>	An interface for sequence style push suppliers. It supports operations for receiving batches of structured events.
<i>StructuredPushConsumer</i>	An interface for structured push consumers.
<i>StructuredPushSupplier</i>	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
<i>Filter</i>	Interface for a filter. The filter supports match operations for the three different event types as well as operations for managing filter constraints.
<i>FilterAdmin</i>	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.
<i>FilterFactory</i>	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
<i>ConsumerAdmin</i>	An interface for consumer administration objects. The Notification Service version of this interface supports the <code>ConsumerAdmin</code> interface from the Event Service as well as the <code>QoSAdmin</code> , <code>NotifySubscribe</code> and <code>FilterAdmin</code> interfaces.
<i>EventChannel</i>	An interface for the event channel. The Notification Service version of this interface supports the <code>EventChannel</code> interface from the Event Service as well as the <code>QoSAdmin</code> and <code>AdminPropertiesAdmin</code> interfaces.
<i>EventChannelFactory</i>	An interface for the event channel factory. The factory supports creation and collection management of event channel objects.
<i>ProxyConsumer</i>	A common base interface for proxy consumers. It extends the <code>QoSAdmin</code> and <code>FilterAdmin</code> interfaces to ensure that all proxy consumers support QoS and filter management.

Table 4 CosNotifyChannelAdmin Interfaces (Continued)

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

- Creating and managing the various proxy objects.
- 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.

Supplemental Information

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
<code>ConnectionReliability</code> (<code>BestEffort/Persistent</code>)	x	x	x	
<code>DiscardPolicy</code> ¹ (<code>Any, FIFO, Priority, Deadline, LIFO</code>)	x	x	x	
<code>EventReliability</code> (<code>BestEffort/Persistent</code>)	x			x
<code>MaxEventsPerConsumer</code> ¹ (<code>long</code>)	x	x	x	
<code>MaximumBatchSize</code> ² (<code>long</code>)	x	x	x	
<code>OrderPolicy</code> (<code>Any, FIFO, Priority, Deadline</code>)	x	x	x	
<code>PacingInterval</code> ² (<code>TimeT</code>)	x	x	x	
<code>Priority</code> (<code>short</code>)	x	x	x	x
<code>StartTime</code> (<code>UtcT</code>)				x
<code>StartTimeSupported</code> (<code>boolean</code>)	x	x	x	
<code>StopTime</code> (<code>UtcT</code>)				x
<code>StopTimeSupported</code> (<code>boolean</code>)	x	x	x	
<code>Timeout</code> (<code>TimeT</code>)	x	x	x	x

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`.

StartTime

The `StartTime` QoS property can only be set in the header of a structured event. It defines the point in time after which the Notification Service is allowed to deliver the event. The start time is an absolute value, where the units are 100 nanoseconds since base time. Base time is defined as 00:00:00 local time, October 15, 1582.

i Proxy objects may be configured to ignore event start times by setting the `StartTimeSupport` QoS property to `FALSE`.

StopTime

The `StopTime` QoS property can only be set in the header of a structured event. It defines the absolute timeout of an event. The Notification Service deletes this event from all queues when timeout occurs. An event that expires from a proxy queue is treated as though it had never been received by the Notification Service. The unit is 100 nanoseconds since base time, where base time is defined as 00:00:00 local time, October 15, 1582.

The event stop time QoS is always applicable. It may be further used 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.

StartTimeSupported

The `StartTimeSupported` QoS property controls whether or not event headers with a start time setting will be processed. The default value for the `StartTimeSupported` QoS is `TRUE`. This QoS can be applied at different levels, for example one proxy object may have start time values supported whereas another proxy has the start times disabled. It is possible to use the `StartTimeSupported` QoS to allow certain privileged consumers to receive events immediately.

StopTimeSupported

The `StopTimeSupported` QoS property controls whether or not event headers with a stop time setting will be processed. The default value for the `StopTimeSupported` QoS is `TRUE`. This QoS applies to both events with a `StopTimeSupported` QoS value and events with a `Timeout` QoS value. It is possible to use the `StopTimeSupported` QoS to allow certain consumers to receive all events, such as for data collection purposes.

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`.

Events spend relatively little time in the event channel before being delivered to the proxy suppliers due to the Notification Service's architecture. In order It is better to use `MaxEventsPerConsumers` on the proxy suppliers rather than `MaxQueueLength` on the event channel in order to effectively apply a discard order.

In general, it is not common for sufficient events to accumulate in the channel to reach `MaxQueueLength`, but setting `MaxQueueLength` is still useful (when used in conjunction with `MaxEventsPerConsumers`) to impose an overall limit on the number of events within the service.

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
<i>MaxReconnectAttempts</i> ¹ (<i>long</i>)	x	x	x	
<i>ReconnectInterval</i> ² (<i>TimeT</i>)	x	x	x	
<i>ConnectedClient</i> ² (<i>Object</i>)			x	
<i>MaxInactivityInterval</i> ³ (<i>TimeT</i>)	x	x	x	
<i>AutoSequenceBatchSize</i> (<i>long</i>)	x	x	x	
<i>AutoSequenceTimeout</i> (<i>ulonglong</i>)	x	x	x	
<i>DisconnectCallback</i>	x	x	x	
<i>MaxMemoryUsage</i>	x			
<i>MaxMemoryUsagePolicy</i>	x			
<i>MemoryCheckInterval</i>	x			
<i>MemoryEscalationExponent</i>	x			
<i>MemoryMaxRecoveryAttempts</i>	x			
<i>MemoryTargetMargin</i>	x			
<i>PropagateQoS</i>	x	x		
<i>DiscardedEvents</i>			x	
<i>DiscardedEventCount</i>			x	

¹This QoS property applies only to proxy push suppliers.

²This QoS property is read only.

³This QoS property applies only to 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 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 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 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.

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 operation 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 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. 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.

The default value is 0 events, which disables auto-sequencing. To enable auto-sequencing, set this QoS to greater than 0 and set `AutoSequenceTimeout` to a value greater than equal to 10. See [“Auto-sequencing”](#) 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 0 milliseconds, which disables auto-sequencing. To enable auto-sequencing, set this QoS to a value greater or equal to 10 and set `AutoSequenceBatchSize` to 1 or greater. See [“Auto-sequencing”](#) 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 be 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*.

DiscardedEvents

The *DiscardedEvents* QoS property provides a mechanism for detecting when a proxy supplier has discarded one or more events: proxy suppliers can set this property to true in order to indicate that at least one event has been discarded.

Setting this property to false indicates that no events have been discarded. The *DiscardedEvents* property can be re-set (to the false value) either by using the supplier proxy's *set_qos()* method or by using the Notification Service Manager's GUI.

Clients are not allowed to set *DiscardedEvent* to true: attempts by a client to do so will be ignored by the QoS (note that the server will not throw an exception if an attempt is made). The *DiscardedEvents* property value is a type boolean.

DiscardedEventCount

The *DiscardedEventCount* is complimentary to the *DiscardedEvents* QoS property. The *DiscardedEventCount* property value is a *long* type (a CORBA *ulonglong*) showing the total number of events which have been discarded. The value cannot be reset: attempts to modify the value will be ignored.

Memory Management Properties

Each event channel has a memory manager. The manager periodically monitors and controls the channel's memory usage. The QoS properties described below are used to set the memory management control parameters and behaviour. Generally, the memory manager keeps memory usage at or below a maximum memory usage level. If this level is exceeded, then it will *attempt* to return the memory usage to a level at or below the desired maximum. Please note that if it may not be possible, under extreme situations, for the system to be kept under the desired maximum memory level.

MemoryCheckInterval

The memory manager checks memory usage at discrete intervals. The *MemoryCheckInterval* property value sets the interval, in milliseconds, between checks. The default value is 5000 milliseconds (five seconds). The property value type is a CORBA `ulonglong` (Java `long`).

A value of 0 milliseconds will cause the memory manager to halt the checking of memory usage. Setting the *MemoryCheckInterval* to a value greater than 0 will cause memory checking to be resumed.

MemoryEscalationExponent

Memory recovery is attempted whenever memory usage exceeds the *MaxMemoryUsage* property value. The memory manager instructs channel components to release memory in this situation, using appropriate methods.

If the component fails to free a sufficient amount of memory using its chosen method, then the manager make another attempt to recover memory by directing the component to free memory by using a more severe method. The manager successively directs a more severe memory recover method each time the component fails to release sufficient memory.

The *MemoryEscalationExponent* property controls the rate of increase of the level of the memory recover method used. The rate of increase is applied exponentially using:

$$n \text{ } ^{\text{EXPONENT}}$$

where

n is the current attempt number (the first attempt is 1, second is 2, etc)
EXPONENT is the exponential value.

The *MemoryEscalationExponent* property sets the value of the *EXPONENT*.

For example, if *MemoryEscalationExponent* is set to 2, the escalation levels will be increased as follows:

first attempt	$1^2 = 2$
second attempt	$2^2 = 4$
third attempt	$3^2 = 8$

The default value is two (2). The property value type is a CORBA `long` (Java `int`).

MemoryMaxRecoveryAttempts

The memory manager can repeatedly direct channel components to free memory whenever the maximum allowed memory usage is reached, as

described above under *MemoryEscalationExponent*: the severity of the memory recovery method increases on each attempt.

However, overall system performance can degrade after the severity level increases beyond a sufficiently high level. There will not be any benefits if memory recovery efforts increase or continue when this situation occurs. The *MemoryMaxRecoveryAttempts* property is provided to stop memory recovery efforts when extreme memory usage situations are reached: CPU resources, which are being used to recover memory, can be returned to the system for processing events. (The term *extreme* in this context indicates a situation where, for example, supplier clients are sending such high numbers of events that the physical limits of the service and system are breached. If extreme conditions are reached more than occasionally, then additional Notification Service resources should be provided, such as providing additional CPUs, federating Notification Service servers across CPUs or hosts, etc. for the number of clients being served.)

This property helps to tune the system for the best balance between performance and memory usage control, as well as protecting the system from dangerous or pointless severity escalation during extreme conditions.

The *MemoryMaxRecoveryAttempts* is disabled if it is set to zero (0), in other words, memory recovery attempts will not be stopped. The default value is ten (10), in other words, memory recover will be escalated up to ten times. The property value type is a CORBA `long` (Java `int`).

MemoryTargetMargin

The memory manager attempts to maintain memory usage at or below level set by the *MaxMemoryUsage* value. When this level is exceeded, the manager directs components to free memory in order to return the memory usage to a level at or below the *MaxMemoryUsage* value.

If usage level is simply returned to the *MaxMemoryUsage* level, but no lower, then it is likely that the maximum will be quickly exceeded again, requiring the manager to release memory again, reducing performance.

The *MemoryTargetMargin* property provides a margin below the *MaxMemoryUsage* value, in bytes, which the memory usage should be freed to when memory is released by the manager. This can prevent calls being immediately made on the manager to release memory and thereby giving the system some breathing space.

No memory margin is provided when the *MemoryTargetMargin* property value is set to zero (0). The default value is 204800 bytes (200K). The property value type is a CORBA `ulonglong` (Java `long`).

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.
- `FALSE` - New events received by the event channel are discarded according to the `DiscardPolicy` QoS setting when the maximum queue length is exhausted. Push 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.

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

Table 7 Notification Service Exceptions (Continued)

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

Part II

Configuration and Management

In this part

This part contains the following:

Notification Service Configuration	page 69
Notification Service Manager	page 95
ChannelConfigurator Tool	page 111

Notification Service Configuration

This section describes the configuration of the Notification Service Singleton properties. These properties appear in the Administration Manager, a graphical user interface (GUI) based administration tool included with the OpenFusion Graphical Tools. In addition to using the Administration Manager to set the Singleton properties, the properties can also be set programmatically.

*Details for configuring Persistence, Logging, CORBA, Java and System properties for the Notification Service are described in the **System 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

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:

```
http://www.microfocus.com/of/servers/NotificationService.ior  
corbaloc::server.microfocus.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

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:

$$\text{Events in queues} + (\text{Events in channel} * \text{Number of proxies})$$

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	DYNAMIC
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

Number of Custom Filters Created

The number of custom filters that have been created using the filter factory since the service was last started.

Property Name	CustomFiltersCreated
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Number of Attached Filters

The number of filters attached to the admins and proxies.

Property Name	AttachedFilters
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Number of Filters Added

The number of times a filter has been added to an admin or proxy.

Property Name	FiltersAdded
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Number of Standard Filters Destroyed

The number of standard filters (that were created using the filter factory) that have been destroyed since the service was last started.

Property Name	StandardFiltersDestroyed
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Number of Standard Filters Created

The number of standard filters that have been created using the filter factory since the service was last started.

Property Name	StandardFiltersCreated
Property Type	DYNAMIC
Data Type	COUNTER

Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Number of Events Rejected by Filters

The number of events rejected by filters.

Property Name	EventsFiltered
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Number of Filters Removed

The number of times a filter has been removed from an admin or proxy.

Property Name	FiltersRemoved
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Number of Buffered Events

The total number of event buffered in the sequence proxy push suppliers.

Property Name	BufferedEvents
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Number of Events Received

The running total of events received from suppliers.

Property Name	EventsReceived
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Minimum Threadpool Size

The minimum number of threads in the thread pool.

Property Name	MinThreads
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Number of Free Threads in the Threadpool

The number of free threads in the thread pool

Property Name	FreeThreads
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Number of Pending Jobs

The number of jobs that are pending execution.

Property Name	PendingJobs
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

Maximum Threadpool Size

The maximum number of threads in the thread pool.

Property Name	MaxThreads
Property Type	DYNAMIC
Data Type	COUNTER

Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

The Number of Working Threads

The number of threads in the thread pool that are executing jobs.

Property Name	WorkingThreads
Property Type	DYNAMIC
Data Type	COUNTER
Accessibility	READ ONLY
Mandatory	NO
getValue() Return Type	longlong

The Number of Current Threads

The number of threads currently in the thread pool.

Property Name	CurrentThreads
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 Oracle HTML Adapter

Checkbox. If this is true (checked), then the Oracle HTML Adapter will be started alongside the service. The Adapter runs for as long as the notification service does.

The Oracle HTML Adapter is a utility provided by Oracle Corporation 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 Oracle HTML Adapter**) and ensure it is started with the service (**JMX Instrumentation: Start Oracle 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 Oracle HTML Adapter

A numeric value which specifies which port the Oracle 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 Oracle HTML Adapter, for example.

Lock Set Factory: Fairness Policy

The fairness policy for the lock set factory. Options are:

- FIFO
- JVM

i

Although **JVM** is shown as an option, it is not currently implemented. **FIFO** will 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

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 is allowed before a transaction times out, in milliseconds. The default timeout is set to 0, which is an unlimited timeout. It is recommended that this value is changed to reflect the specific needs of the system. For example, moderately loaded systems might use a value of 60000 (60 seconds); a heavily loaded system needed a higher value or may even retain the default unlimited timeout value.

Property Name	components/TransactionManager/domain/timeout
Property Type	FIXED
Data Type	INTEGER
Accessibility	READ/WRITE
Mandatory	NO

Event Database: Purge Rate

When using file persistence for the service, 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

When using file persistence, 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 Synchronisation

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). This property only applies to file persistence.

The default value of this property is `false`.

Property Name	components/Journal/guaranteedsyncing
Property Type	STATIC
Data Type	BOOLEAN
Accessibility	READ/WRITE
Mandatory	YES

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.microfocus.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 Object Hierarchy](#) in 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

Notification Service Manager

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.

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 Event Channels have been created programmatically, the manager will display the default service `EventChannelFactory` object below the Notification Service icon itself (Figure 10).

If the **ChannelConfigurator** Object is present, a saved configuration may be loaded.

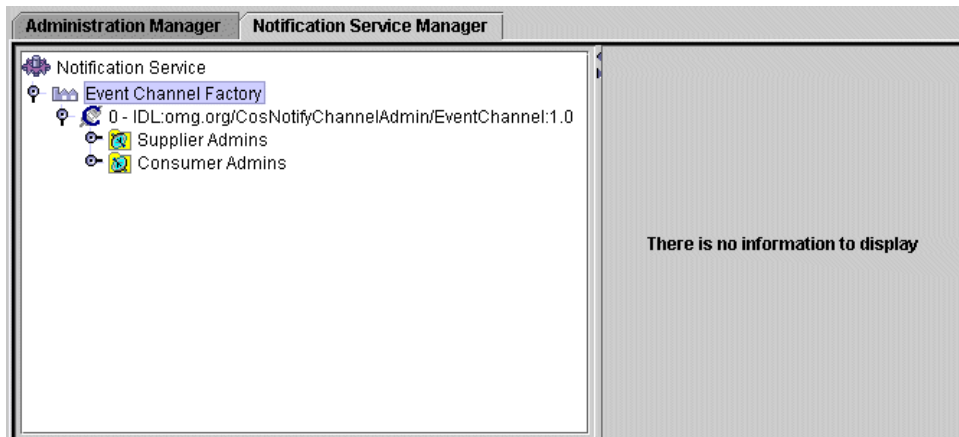


Figure 10 Notification Service Manager

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 8](#).

Table 8 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.
	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.

Table 8 Notification Service Nodes (Continued)

Icon	Object
	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

- 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*.
- 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”](#) 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 [“Quality of Service Properties”](#) 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 11](#).

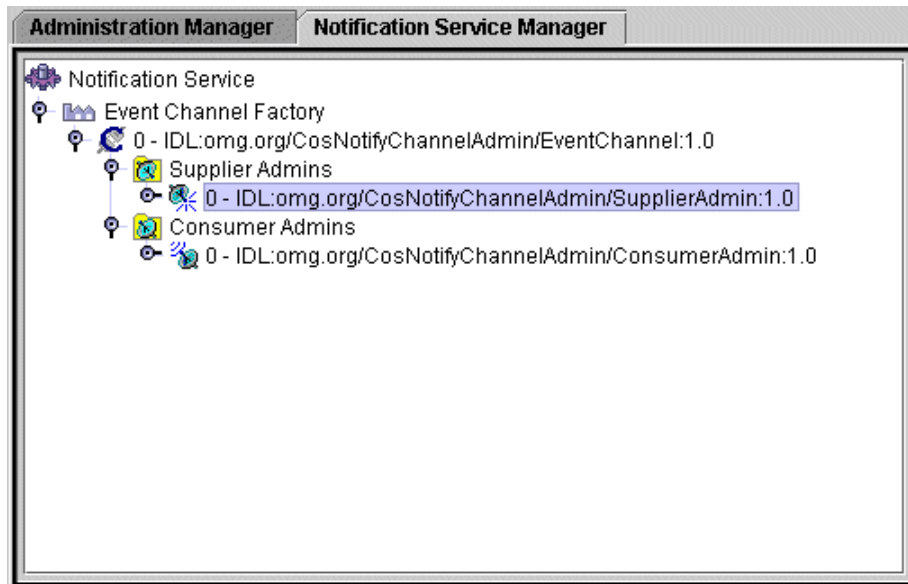


Figure 11 Supplier and Consumer Admins

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, displaying the **QoS Settings** associated with the object.

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 [“Quality of Service Properties”](#) 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 12](#).

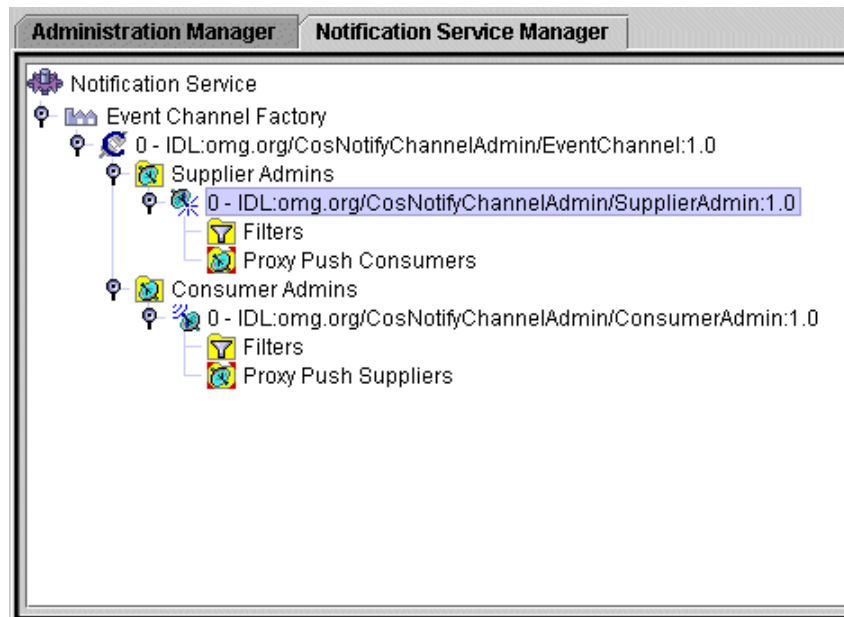


Figure 12 Filters

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

- 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 13](#).

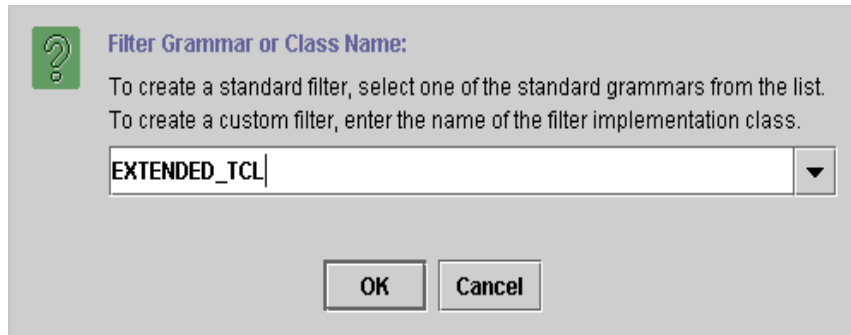


Figure 13 Add Filter

- 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.
- 3 Click the **OK** button.
- 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 14](#).

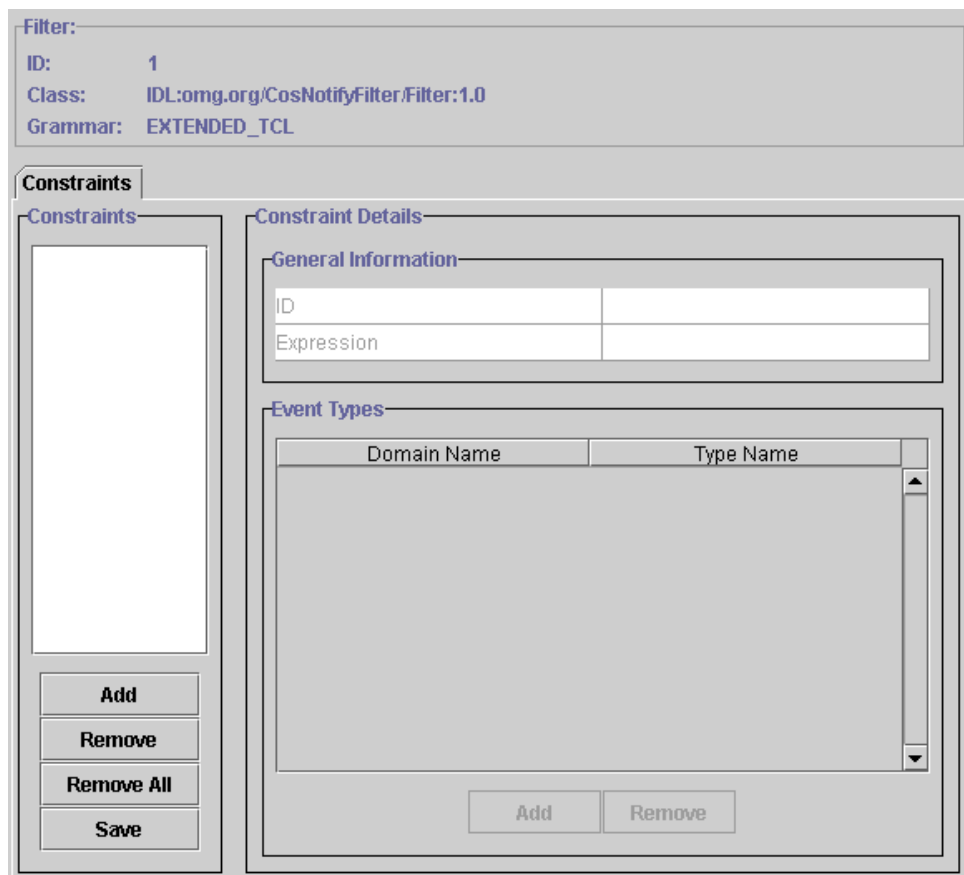


Figure 14 Filter Details

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

- 5 Click the **OK** button once the full constraint expression has been entered.
- 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

- 1 To remove a constraint, select the constraint in the **Constraints** list.
- 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 16](#). 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.

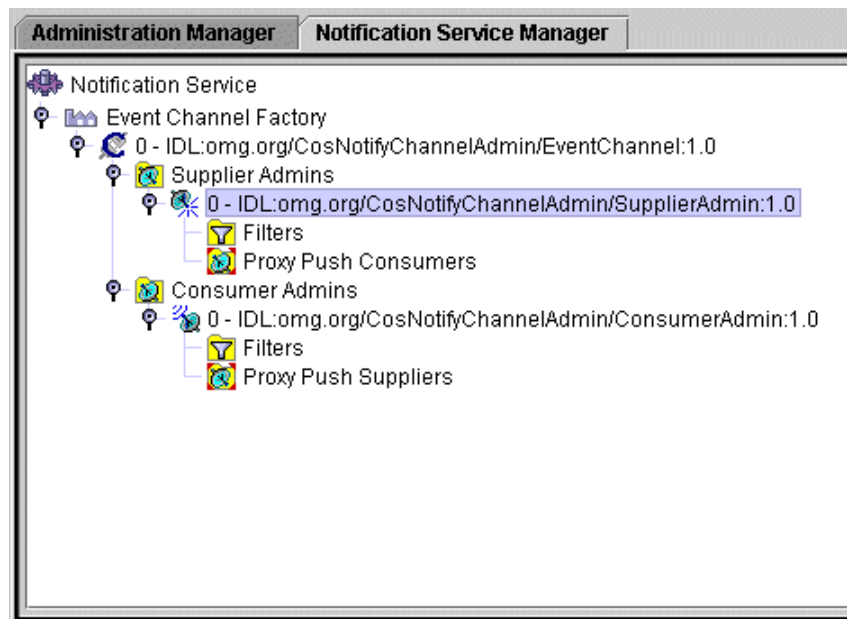


Figure 16 Proxy Objects

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 [“Quality of Service Properties”](#) 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.

- 1 To create a new Proxy Object, select the relevant node in the Notification browser hierarchy pane:
 - a Proxy Push Supplier
 - b Proxy Push Consumer
- 2 Right-click on the line in the hierarchy tree and select the **Obtain New Proxy** option from the pop-up menu.
- 3 Select the Client Type from the list box: **Structured**, or **Sequence**.
- 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”](#) 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”](#).
- 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 17](#) 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.

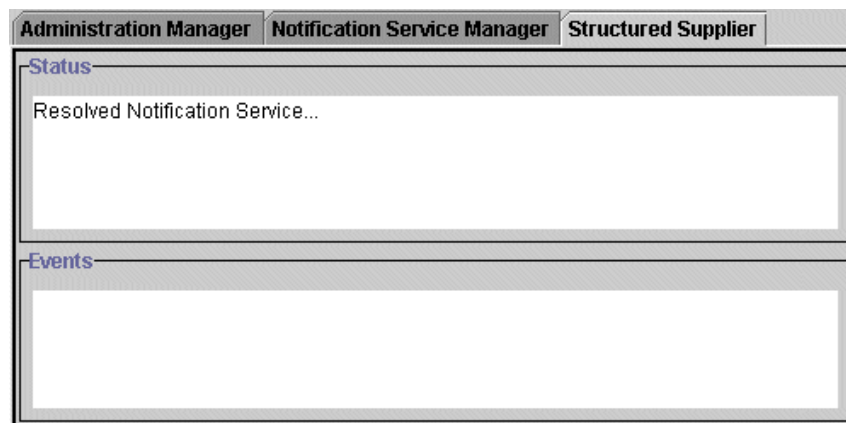


Figure 17 Structured Supplier Manager

Configuring the Structured Consumer

Figure 18 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.

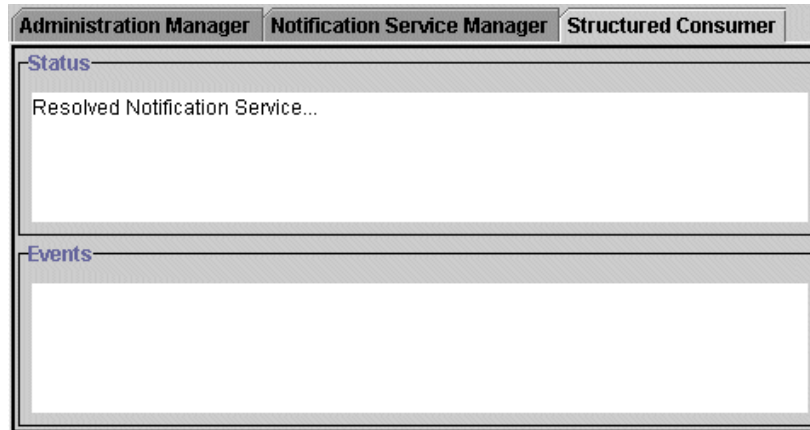


Figure 18 Structured Consumer Manager



The textual representations of events sent and received by the Test Client GUIs will take up space in memory while they are displayed (as all text does within any text pane). The user should be aware that this could potentially cause memory exhaustion in the Administration Manager process if messages are sent or received over extended periods.

Connecting the Structured Supplier

When the Structured Supplier Manager is invoked, the Structured Supplier client resolves the Notification Service.

- 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.
- 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.

- 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.
- 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.

- 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 19](#).

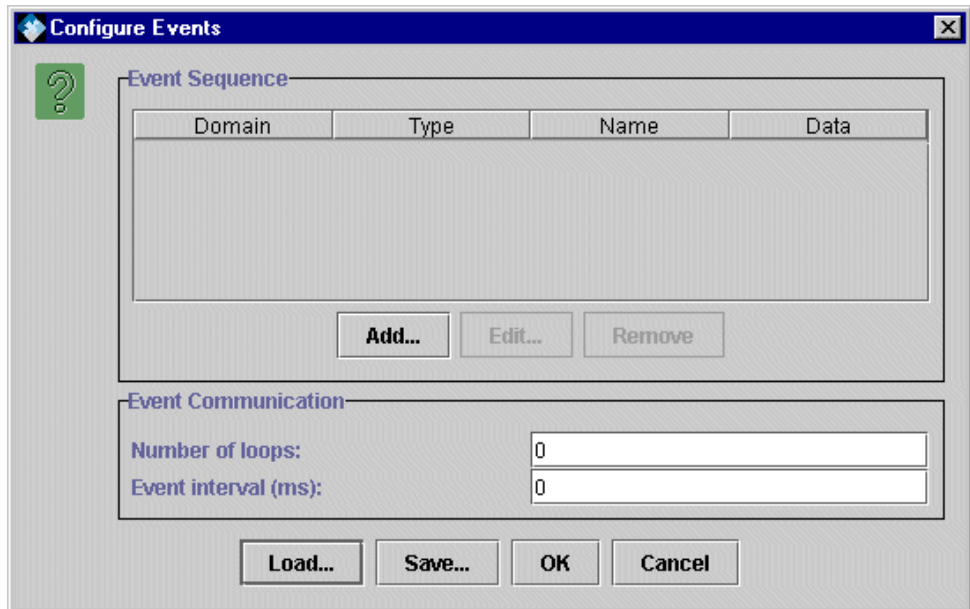


Figure 19 Configure Events Dialog Box

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.

- 1 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.
- 2 Click the **Add** button in the **Event Sequence** pane. This gives a dialog box for creating structured events, shown in [Figure 20](#).

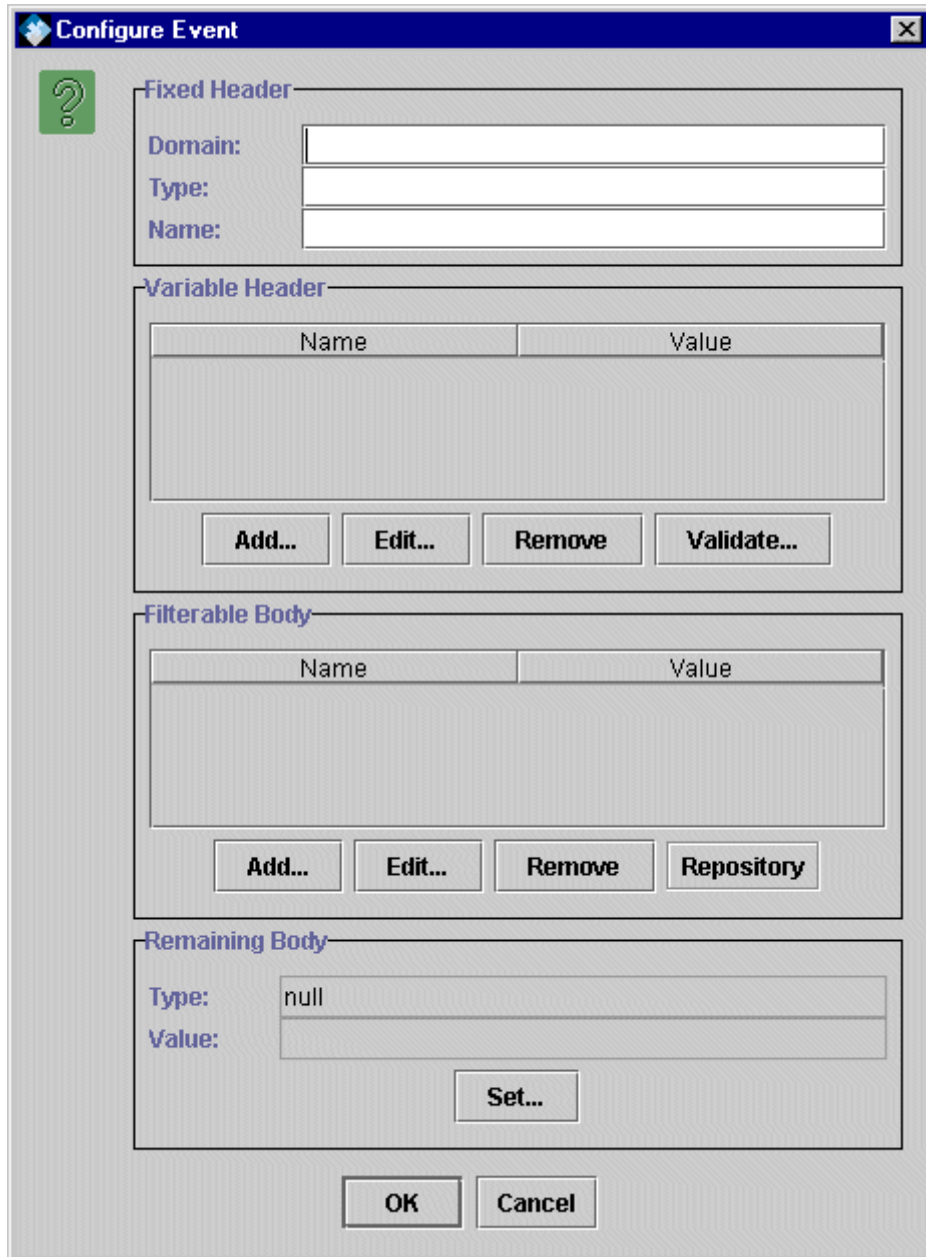


Figure 20 Configure Event Dialog Box

- 3 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`).
- 4 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.
- 5 Click **OK** to load the event into the **Event Sequence** table of the **Configure Events** dialog.
- 6 Repeat step 3 through step 6 as before, but give this event a different identifier and set the severity to `4`.

- 7 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

- 1 To begin transmitting the events, click the **Send Events** button on the tool bar.
- 2 If you examine the Structured Supplier Manager you should notice the events being transmitted in the Events pane.
- 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.

- 1 Select the Notification Service Manager window and create a new Filter object on the Supplier Admin object.
- 2 Create a new constraint.
- 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.
- 4 Clear the **Events** panes in the Structured Supplier and Consumer Manager windows and click the **Send Events** button again.
- 5 Examine the **Events** pane in the Structured Supplier Manager. Both events are transmitted to the Event Channel.
- 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.

ChannelConfigurator Tool

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.

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

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.



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.

When the channel configurator writes the time-related QoS property values (*MaxInactivityInterval*, *PacingInterval*, *ReconnectInterval*, *ThreadIdleTime* and *Timeout*) to the XML file, it changes the units from 100 nanoseconds to milliseconds. When the configurator reads in the XML file to recreate the service configuration, it will convert the values back to 100ns units.

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 21.

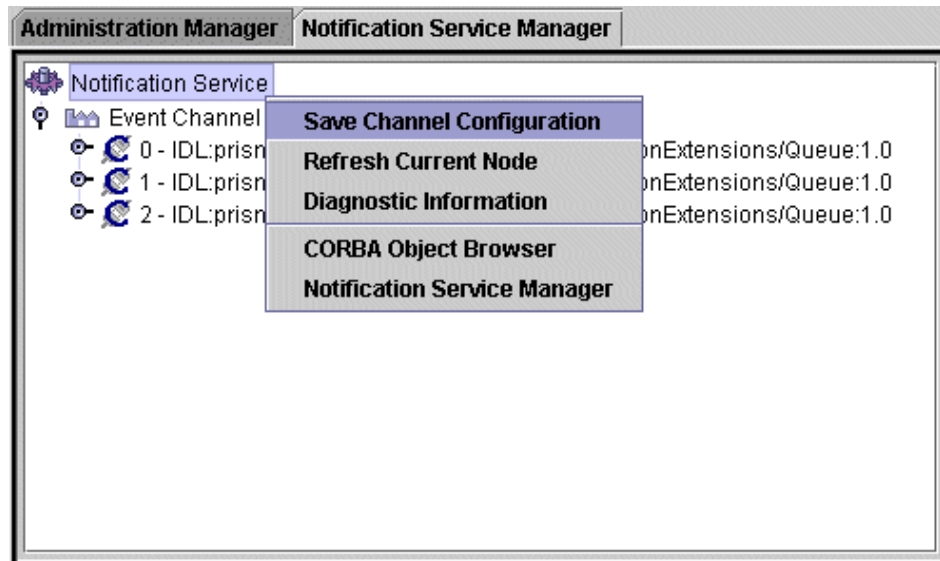


Figure 21 Saving Channel Configuration

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.

Running from the Command Line

To load a saved channel configuration into the Notification Service:

```
% run com.prismt.cos.CosNotification.tools.config.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.config.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

A

- Adding
 - Constraints 102
- Admin Objects 13
- Admin Properties 98
- Administration Interfaces 51
- AutoSequenceBatchSize (QoS property) 59, 98
- AutoSequenceTimeout (QoS property) 60, 98
- Auto-sequencing 18

B

- Blobstore Logger Level (property) 76

C

- Channel
 - Configuration 111
- Channel Configuration URL (property) 112
- ChannelConfigurationURL (property) 112
- ChannelConfigurator 96, 111
- Channels 96
- Channels (property) 83
- Component
 - Connection 9
 - Creation 9
- Component Manager Logger Level (property) 79
- components/EventDatabase/
 - maxpurgememory (property) 91
- components/Journal/guaranteedsyncing (property) 91
- components/LockSetFactory/fairness (property) 89
- components/ThreadPool/pool-initial (property) 89
- components/ThreadPool/pool-max (property) 89
- components/ThreadPool/pool-min (property) 89
- components/ThreadPool/thread-timeout (property) 90
- components/TransactionManager/domain/
 - timeout (property) 90
- Configuring a Structured Supplier 105
- ConnectedClient (QoS property) 59
- ConnectionReliability (property) 54
- ConnectionReliability (QoS property) 98, 99, 104
- Constraint Language 17
- Constraints
 - Adding 102
 - Removing 103
- Consumer Admin 96
- Consumer Admins 96
 - Setting up 98
- ConsumerAdmins (property) 81
- create_channel Operation 49
- Creating
 - a New Filter 100

- Test Events 107
- Current Total of Events Awaiting Delivery (property) 82
- Current Total of Events in Channels (property) 81
- CurrentEvents (property) 81

D

- Database Plugin Class (property)
 - Notification Service 70
- DB.Plugin (property)
 - Notification Service 70
- DB.WAL (property) 69
- DB.WAL.Dir (property) 70
- DB.WAL.MaxSize (property) 70
- default_consumer_admin Operation 50
- default_filter_factory Operation 50
- default_supplier_admin Operation 50
- Dependencies (on Other Services) 7
- DiscardPolicy (QoS property) 56
- DisconnectCallback (QoS property) 60, 104
- documentation
 - .pdf format ix
 - updates on the web ix

E

- Enable Write Ahead Log (property) 69
- Errors 65
- Event
 - Body 12
 - Communication Models 12
 - Header 11
 - Transmission 8
- Event Channel 12
 - Factory 96
 - Properties 97
 - Setting up 97
- Event Channel Factory
 - create_channel Operation 49
- Event Channel Factory Interface 49
- Event Channel Interface
 - default_consumer_admin Operation 50
 - default_filter_factory Operation 50
 - default_supplier_admin Operation 50
 - destroy Operation 50
 - for_consumers Operation 50
 - for_suppliers Operation 50
 - get_admin Operation 50
 - get_all_consumeradmins Operation 50
 - get_all_supplieradmins Operation 50
 - get_consumeradmin Operation 50
 - get_qos Operation 50
 - get_supplieradmin Operation 50
 - MyFactory Operation 50
 - new_for_consumers Operation 50
 - new_for_suppliers Operation 50
 - set_admin Operation 50
 - set_qos Operation 50
- Event Database

- Maximum Purge Memory (property) 91
- Purge Rate (property) 90
- EventChannelFactory Object 95
- EventReliability (QoS property) 53, 98
- Events Delivered (property) 81
- Events Received (property) 80
- Events, Defined 11
- Events, Structured 11
- EventsAwaitingDelivery (property) 82
- EventsDelivered (property) 81
- EventsReceived (property) 80
- Exceptions 65

F

- Federation 20
- Filter 96
 - Events 109
 - Interfaces 51
 - Removing 103
- Filtering 16
- for_consumers Operation 50
- for_suppliers Operation 50

G

- get_admin Operation 50
- get_all_consumeradmins Operation 50
- get_all_supplieradmins Operation 50
- get_consumeradmin Operation 50
- get_qos Operation 50
- get_supplieradmin Operation 50
- getValue() method 80
- GlobalSetting (property) 73

I

- Instrumentation
 - Notification Service Properties 80
- Instrumentation Properties 80
- IOR File Name (property) 71, 92
- IOR Name Service (property) 71, 93
- IOR Name Service Entry (property) 71, 92
- IOR URL (property) 71, 92
- IOR_URL Element 112
- IOR.File (property) 71, 92
- IOR.URL (property) 71, 92

J

- JMX (Instrumentation) Properties 80
- Journal
 - Guaranteed Syncing (property) 91
- JTO Logger Level (property) 74

L

- Local Channel 21
- Lock Set Factory
 - Fairness Policy (property) 89
- Lock Set Factory Logger Level (property) 79
- logcategory/blobstore (property) 76
- logcategory/ecfc (property) 78
- logcategory/ecm (property) 79
- logcategory/jto (property) 74

- logcategory/locksetfactory (property) 79
- logcategory/messenger (property) 75
- logcategory/orb (property) 75
- logcategory/rolemanager (property) 74
- logcategory/scheduler (property) 73
- logcategory/statefactory (property) 77
- logcategory/statemachinefactory (property) 77
- logcategory/threadpool (property) 78
- logcategory/transactionmanager (property) 76
- logkit/targets/file/filename (property) 72
- logkit/targets/file/format (property) 73

M

- Managing
 - Proxies 28, 33
- MaxConsumers (admin property) 64, 98
- MaxEventsPerConsumer (QoS property) 55, 98, 99, 104
- Maximum Queue Size (property) 88
- MaximumBatchSize (QoS property) 56, 98, 99, 104
- MaxInactivityInterval (QoS property) 58
- MaxMemoryUsage (QoS property) 60
- MaxMemoryUsagePolicy (QoS property) 60
- MaxQueueLength (admin property) 64, 98
- MaxQueueSize (property) 88
- MaxReconnectAttempts (QoS property) 58, 98, 99, 104
- MaxSuppliers (admin property) 64, 98
- Messenger Logger Level (property) 75
- MyFactory Operation 50

N

- NameServiceName (property) 111
- new_for_consumers Operation 50
- new_for_suppliers Operation 50
- Notification Service
 - Configuration 69
 - Errors 65
 - Event Channel Factory, create_channel Operation 49
 - Event Channel Interface
 - default_consumer_admin Operation 50
 - default_filter_factory Operation 50
 - default_supplier_admin Operation 50
 - destroy Operation 50
 - for_consumers Operation 50
 - for_suppliers Operation 50
 - get_admin Operation 50
 - get_all_consumeradmins Operation 50
 - get_all_supplieradmins Operation 50
 - get_consumeradmin Operation 50
 - get_qos Operation 50
 - get_supplieradmin Operation 50
 - MyFactory Operation 50
 - new_for_consumers Operation 50

- new_for_suppliers Operation 50
- set_admin Operation 50
- set_qos Operation 50
- Exceptions 65
- Hierarchy 96
- Introduction 5, 23, 47, 53
- Manager 95
- Proxy Management 28, 33
- Quality of Service Property
 - ConnectedClient 59
 - ConnectionReliability 54
 - DiscardPolicy 56
 - EventReliability 53
 - MaxEventsPerConsumer 55
 - MaximumBatchSize 56
 - MaxInactivityInterval 58, 59
 - MaxReconnectAttempts 58
 - OrderPolicy 56
 - PacingInterval 56
 - Priority 54
 - ReconnectInterval 59
 - StartTimeSupported 55
 - StopTime 54
 - StopTimeSupported 55
 - Timeout 55
- Service Dependencies 7
- Notification Service Logger Level (property) 78
- NotificationServiceName (property) 111
- NotificationSingleton Configuration 69
- NS_Entry Element 112
- Number of Consumer Admins (property) 81
- Number of Event Channels (property) 83
- Number of Proxy Push Consumers (property) 80
- Number of Proxy Push Suppliers (property) 82
- Number of Sequence Proxy Push Consumers (property) 81
- Number of Sequence Proxy Push Suppliers (property) 83
- Number of Structured Proxy Push Consumers (property) 80
- Number of Structured Proxy Push Suppliers (property) 82
- Number of Supplier Admins (property) 83

O

- Object.Name (property) 71, 92
- OMG
 - Standard API Definitions 47
 - Standard Features 5
- OpenFusion
 - Enhancements 6
 - QoS Extensions 16, 58
- ORB Logger Level (property) 75
- OrderPolicy (QoS property) 56, 98, 99

P

- PacingInterval (QoS property) 56, 98, 99, 104
- Passivating Persistent Clients 20
- PDF documentation ix
- Persistence 19
- Priority 54
- Priority (QoS property) 54, 98, 99, 104
- Process.getValue() 80
- ProcessSingleton Configuration
 - Notification Service 92
- PropagateQoS (QoS property) 61, 98
- Proxy
 - Defined 13
 - Instances 103
 - Management 28, 33
 - Push Consumers 97
 - Push Suppliers 97
- Proxy Objects
 - Destroying 109
- Proxy Push Consumer 97
- Proxy Push Supplier 97
- ProxyPushConsumers (property) 80
- ProxyPushSuppliers (property) 82

Q

- QoS Settings 99
 - Proxy Objects 104
- Quality of Service Property
 - ConnectedClient 59
 - ConnectionReliability 54
 - DiscardPolicy 56
 - EventReliability 53
 - MaxEventsPerConsumer 55
 - MaximumBatchSize 56
 - MaxInactivityInterval 58, 59
 - MaxReconnectAttempts 58
 - OrderPolicy 56
 - PacingInterval 56
 - Priority 54
 - ReconnectInterval 59
 - StartTimeSupported 55
 - StopTime 54
 - StopTimeSupported 55
 - Timeout 55
- Queues, Defined 14

R

- Reconnecting Consumers (property) 83
- ReconnectingConsumers (property) 83
- ReconnectInterval (QoS property) 59, 98, 99, 104
- RejectNewEvents (admin property) 98
 - Notification Service 64
- Removing
 - Constraints 103
 - Filters 103
- Requirements 19
- Resolve Name (property) 72
- ResolveName (property) 72
- Resuming Connections 13

Role Manager Logger Level (property) 74

S

SequenceProxyPushConsumers
(property) 81

SequenceProxyPushSuppliers
(property) 83

Sequencing 17

Service Log File Format (property) 72

Service Log File Location (property) 72

Set All Loggers To (property) 73

set_admin Operation 50

set_qos Operation 50

Singletons

- NotificationSingleton 69

Standard

- OMG Properties 15, 53

Starting the Notification Service
Manager 95

StartTime 54

StartTimeSupported (QoS property) 55

State Factory Logger Level (property) 77

State Machine Factory Logger Level
(property) 77

StopTime (property) 54

StopTimeSupported (QoS property) 55

Structured Consumer, Connecting 106

Structured Events 11

Structured Supplier, Configuration 106

StructuredProxyPushConsumers
(property) 80

StructuredProxyPushSuppliers
(property) 82

Supplier Admin 96

Supplier Admins 96

- Setting up 98

SupplierAdmins (property) 83

Suspending Connections 13

T

Thread Pool

- Initial Pool Size (property) 89
- Maximum Pool Size (property) 89
- Minimum Pool Size (property) 89
- Thread Timeout (property) 90

Thread Pool Logger Level (property) 78

Timeout (QoS property) 55, 98, 99, 104

Transaction Manager

- Domain Timeout (property) 90

Transaction Manager Logger Level
(property) 76

Transmitting Test Events 109

W

Write Ahead Log 69

Write Ahead Log Directory (property) 70

Write Ahead Log Maximum Size
(property) 70