

OpenFusion® CORBA Services

Version 4.2

Log Service



OpenFusion® CORBA Services

LOG SERVICE GUIDE



Part Number: OFCOR-LOGG-42

Doc Issue 05, 1 June 2005

Notices

Copyright Notice

© 2005 PrismTech Limited. All rights reserved.

This document may be reproduced in whole but not in part.

The information contained in this document is subject to change without notice and is made available in good faith without liability on the part of PrismTech Limited or PrismTech Corporation.

All trademarks acknowledged.

Preface

About the Log Service Guide

The *Log Service Guide* is included with the OpenFusion Log Service' *Documentation Set*. The *Log Service Guide* explains how to use the OpenFusion Log Service.

The *Log Service Guide* is intended to be used with the *System Guide*, *Notification Service Guide*¹, and other OpenFusion Log Service documents included with the product distribution; refer to the *Product Guide* for a complete list of documents.

Intended Audience

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

Organisation

The *Log Service Guide* is organised into two main sections. The first section describes the OpenFusion Log Service. This section provides:

- a high level description and list of main features
- explanation of the 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 service

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

1. When using the *NotifyLog* type.

Conventions

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



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



Item contains helpful hint or special information.

WIN

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

UNIX

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

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

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

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

`Courier` fonts indicate programming code and file names.

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

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

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

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

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

Contacts

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

Corporate Headquarters

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

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

Web: <http://www.prismtechnologies.com>
General Enquiries: info@prismtechnologies.com

European Head Office

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

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

A close-up, slightly blurred photograph of a computer keyboard, viewed from an angle. The keys are light-colored, and the overall image has a soft, ethereal quality. A white grid pattern is overlaid on the entire image, creating a sense of structure and depth. The word "Contents" is centered in a dark blue, serif font.

Contents

Table of Contents

	Notices	v
	Preface	vi
	About the Log Service Guide	vi
	Contacts	viii
	List of Figures	xv
	List of Tables	xvii
	Log Service	1
<i>1</i>	Description	3
<i>1.1</i>	Overview	3
	Benefits	3
	OMG Standard Features	4
	OpenFusion Features	4
	Instrumentation	5
	Dependencies on Other Services	5
<i>1.2</i>	Concepts and Architecture	6
	Basic Architecture	6
	Log Types	6
	Log Factories	9
	Log Networks	11
	Components and Features	11
	Log Records	11
	Log Generated Events	12
	Lifecycle Operations	13
	Log Management	14
	Quality of Service	17
	Log Features Comparison	18
<i>2</i>	Using Specific Features	19
<i>2.1</i>	Introduction	19
	Import statements	20

2.2	The Basics and the BasicLog	21
	Creating a BasicLog	21
	Import Statements	21
	Initialisation	22
	Log Creation and Configuration	22
	Locating the Log	24
	BasicLog Exceptions	25
	Creating a Client for the BasicLog	25
	Import Statements	26
	Client Initialisation	26
	Preparing the Data	27
	Sending the Data	27
	Querying the Log Records	29
	Client Exceptions for the BasicLog	31
2.3	The NotifyLog and Event-style Events	32
	Creating a NotifyLog	32
	Import Statements	32
	NotifyLog Initialisation	33
	NotifyLog Creation and Configuration	33
	Locating the NotifyLog	36
	NotifyLog Exceptions	36
	Creating a Supplier Client for the NotifyLog	37
	Import Statements	38
	Supplier Client Initialisation	38
	The Client as an Event Supplier	40
	Sending Events	41
	Terminating the Supplier Client	42
	Supplier Client Exceptions	43
	Creating a Consumer Client for the NotifyLog	43
	Import Statements	44
	Consumer Client Initialisation	44
	The Client as an Event Consumer	46
	Retrieving Records	47
	Cleanup	48
	Exceptions	48
2.4	NotifyLog, Structured Events and More	48
	Creating a NotifyLog	49
	Import Statements	49
	NotifyLog Initialisation	50
	NotifyLog Creation and Configuration	50
	Filtering	52

Registering with the Naming Service	56
NotifyLog Exceptions	56
Creating a Supplier Client for the NotifyLog	57
Import Statements	57
Supplier Client Initialisation	58
Structured Events in Brief	60
Creating and Sending the Events	61
Supplier Client Exceptions	63
Creating a Consumer Client for the NotifyLog	64
Import Statements	64
Consumer Client Initialisation	65
Setting a Proxy Filter	66
Receiving the Structured Events	67
Retrieving Log Records	68
Cleanup	69
Consumer Client Exceptions	69
Monitoring Log Generated Events	70
Import Statements	70
Monitor Client Initialisation	71
Receiving the Generated Events	71
Monitor Client Exceptions	72
3 Supplemental Information	73
3.1 Exceptions	73
Configuration and Management	75
4 Log Service Configuration	77
4.1 Overview	77
Common Properties	77
4.2 LogFactorySingleton Configuration	77
CORBA Properties	78
Persistence Properties	79
General Properties	81
4.3 ProcessSingleton Configuration	84
5 Log Service Manager	87
5.1 Overview	87

5.2	Using the Log Service Manager	87
	Log Object Settings	88
	Log	88
	Alarms	89
	Scheduling	90
	Miscellaneous	90
	Channel	91
	Events	92
	Create a New Log	93
	Copy Logs	94
	Destroy Logs	94
	Browser	94
	Settings	95
	Operations	95
	Appendices	99
	A Locating a Log with the Naming Service	101
	Creating a Name Binding	101
	Obtaining the Object with Resolve	102
	B Using with JacORB	105
	Index	107

List of Figures

Figure 1	Log Types and Inheritance	7
Figure 2	Basic Architecture of Event Channel-based Log Types	8
Figure 3	Basic Architecture of Notify Channel-based Log Types	9
Figure 4	Log Factory Types and Inheritance	10
Figure 5	Log Generated Event Transmission	11
Figure 6	Structured Event	61
Figure 7	Log Service Manager	87
Figure 8	Viewing Log Details	94

List of Figures

List of Tables

Table 1 Settings Generating AttributeValueChange Events	17
Table 2 Features of Different Log Types	18
Table 3 Log Service Exceptions	73
Table 4 Log Service Object Icons	88

List of Tables

A close-up, low-angle view of a computer keyboard, showing several keys in detail. The keys are white and set against a dark background. A white grid pattern is overlaid on the entire image, creating a sense of depth and perspective. The text "Log Service" is centered in the upper half of the image.

Log Service

1 Description

1.1 Overview

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

The service can be used stand-alone or with other OpenFusion Log Service' interfaces and services.

OpenFusion Log Service is standards based, i.e. fully compliant with recognised industry standards and specifications, and supports portability and interoperability, and conforms with the OMG's *Telecom Log Service Specification* (which is, itself, an extension of the telecommunication industry's *ITU-T X.735* specification for logging events).

The *Telecom Log Service* can log and query any type of Notification or Event Service-based event, including Event Style, Structured, and Typed events. The Telecom Log Service can also log data received from clients which have no knowledge of events.

The OpenFusion Log Service conforms to the OMG's Telecom Log Service Specification (which is, itself, an extension of the telecommunication industry's *ITU-T X.735* specification for logging events). The OpenFusion Log Service supports the Basic and Notify Log functionality of the OMG specification.

i This version of the OpenFusion Log Service is a subset of the Telecom Log Service specification and *only* supports Basic and Notify Logs using *push* mode.

Benefits

Some of the benefits of using the Log Service include:

- the ability to store and selectively query events and *Any* data
- the performance, scalability and control benefits of an event channel architecture, including decoupled asynchronous and synchronous event transmission
- ease of maintenance when adding or removing event suppliers and consumers
- ease of integration and use with the Notification Service
- efficient use of network bandwidth between the suppliers and consumers

The OpenFusion Log Service is widely used in the telecommunications, finance, transport, travel and energy industries for applications which require logging of client generated events, such as faults, errors, or alarms. The Log Service can be used for storing and retrieving a variety of information in a central location, such as:

- audit trails of alarms for Operational Support Systems (OSS)
- audit trails of diagnostics
- configuration and management information

OMG Standard Features

This release of the OpenFusion Log Service includes the standard OMG features, such as:

- the ability to selectively store and query any Notification Service-based events
- the ability to store data, sent as *AnyS*, from clients which have no knowledge of events
- decoupling event transmission from suppliers to consumers by using *event channels* and *proxies*. The events may be *unstructured* (simple data), *structured* (containing details about the event)
- push event communication model for basic events
- leverage of the Notification Service's Quality of Service (QoS) support
- support for multiple suppliers and consumers
- provision of filters and the *Extended Trader Constraint Language* for controlling event transmission to and from clients using the service and for controlling which events are saved to the persistent store
- the ability to forward events to consumers (such as applications or other *log objects*), which can be used to create log networks with *log-and-forward* ability
- the ability to report (using self generated events) when the state or condition of a log changes

OpenFusion Features

The OpenFusion Log 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
- support for persisting events to commercial databases
- provisions for improved performance and scalability, such as
 - multi-threading
 - provision of persistence for events, channels and connections to selected commercial databases through the use of optimized stored procedures
 - automatic service activation on demand
- provision of service monitoring functions, *instrumentation* (described below)

Instrumentation

Log Service provides both general and service-specific instrumentation features which can be used for system monitoring, which in turn aids in problem identification, performance tuning, etc. Log Service Instrumentation consists of a set of properties that can be monitored either using the Administration Manager (one of the Log Service Graphical Tools) or remotely using SNMP.

In addition to properties that are read-only at run time, OpenFusion provides some properties that can be set and reset at run time as required, such as when a particular threshold value is reached or a time period has elapsed. Note that there is virtually no performance overhead involved in using any of the OpenFusion Instrumentation features.

Dependencies on Other Services

The Log Service needs the OpenFusion Notification Service to be installed, although it is not necessary to license the Notification Service (in order to use the Log Service).

The Log Service examples included in this guide use the OpenFusion Naming Service. Although the Log Service itself does not require this service, the Naming Service is highly recommended.

1.2 Concepts and Architecture

There are numerous situations where it is necessary or desirable to be able to dynamically store information about a running system and its components: it may not be sufficient for a system to simply respond to events which occur. It may also be necessary to have a record of those events. The Log Service enables records of events to be stored which can later be retrieved for analysis.

Basic Architecture

The main component of the Telecom Log Service is the *log* or *log object*. Log objects store events and data as log records. Logs can transmit events from suppliers to consumers, like event or notification channels do. Logs can also report their own state or condition through the generation and transmission of their own events.

There are different types of log. All but one of these types can forward events to consumer clients and generate their own events. Further, logs allow clients, which have no knowledge of events or notifications, to store *Any*-based data directly to the log's persistent store.

Logs can also:

- be linked together as networks, similar to the federation of event channels
- support the *copy* and *destroy* Lifecycle operations

Logs are created with *log factories*. Log factories also perform other functions, such as transmitting log generated events to clients (when required).

Log Types

The Telecom Log Service Specification defines five log types. The OpenFusion implements two of these, the *BasicLog* and *NotifyLog*. All log types are inherited from an abstract `Log` interface. Four of the types also inherit from one of four event channel types (Event, Typed Event, Notify, or Typed Notify).



This release of the OpenFusion Log Service only supports *Basic* and *Notify* Logs (described below): descriptions of the other log types as supported by the full Telecom Log Service Specification are included for information and general understanding.

The five log types are:

1. *BasicLog* - allows *event unaware* clients to store *Anys* directly to the log's persistent store. BasicLogs do not support event forwarding nor can they generate events which report the log's state. BasicLogs only inherit from the *Log* interface.
2. *EventLog* - receives, saves to its persistent store, and forwards untyped *Event Style* events. *EventLog* can generate its own events which report the log's state. *EventLog* inherits from both *Log* and *CosEventChannelAdmin::EventChannel*.
3. *TypedEventLog* - receives, saves to its persistent store, and forwards typed *Event Style* events (*Anys*). *TypedEventLog* can generate its own events which report the log's state. *TypedEventLog* inherits from both *Log* and *CosTypedEventChannelAdmin::TypedEventChannel*.
4. *NotifyLog* - receives, saves to its persistent store, and forwards untyped events (including structured events). *NotifyLog* can filter events and generate its own events which report the log's state. *NotifyLog* inherits from both *EventLog* and *CosNotifyChannelAdmin::EventChannel*.
5. *TypedNotifyLog* - receives, saves to its persistent store, and forwards typed events. *TypedNotifyLog* can generate its own events which report the log's state and filter events. *NotifyLog* inherits from both *TypedEventLog* and *CosTypedNotifyChannelAdmin::TypedEventChannel*.

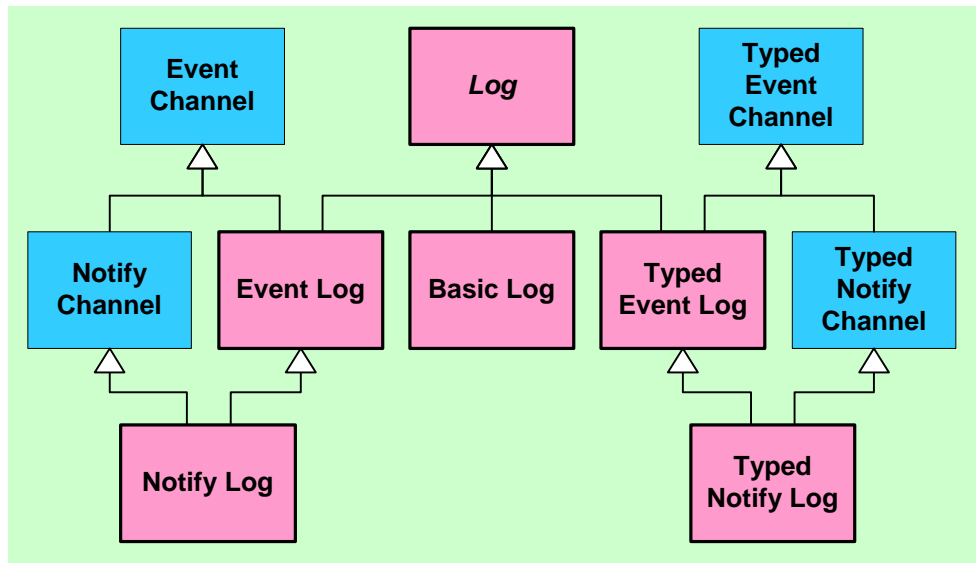


Figure 1 Log Types and Inheritance

All logs, except for the `BasicLog`, inherit the architectural components of event channels. *Figure 2* and *Figure 3* show the basic architectures for event channel-based and notification channel-based logs, respectively.

i

The `Log` inheritance hierarchy, shown in *Figure 1*, allows clients to access a log directly or through one of their underlying services, namely Event Service, Typed Event Service, Notification Service, and Typed Notification Service.

Further, a client can ‘widen’ a log to any of its bases classes. For example, a `NotifyLog` could be widened to a `Log` for use with event unaware applications.

A summary of the different features for each log type is listed in Table 2, *Features of Different Log Types*, on page 18.

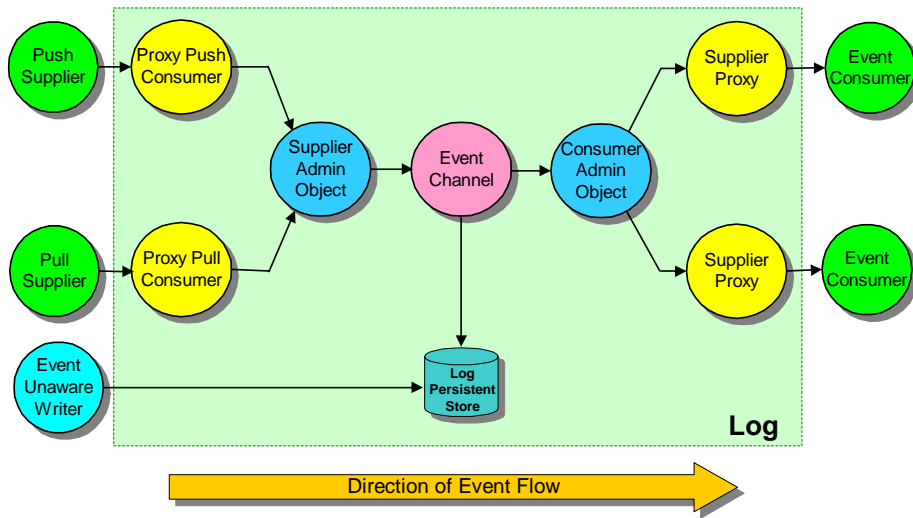


Figure 2 Basic Architecture of Event Channel-based Log Types

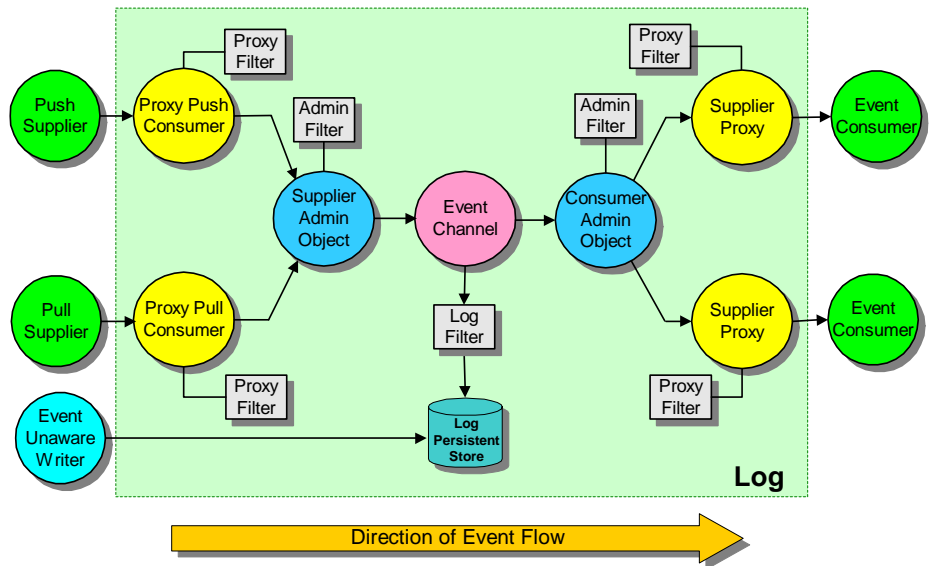


Figure 3 Basic Architecture of Notify Channel-based Log Types

Log Factories

Logs are created by log factories. Log factories perform several functions, including:

- log creation
- performing as a log collection manager
- generating log events
- transmitting log events to subscribed consumers

Log factory interfaces are inherited from the *LogMgr* interface plus either a *ConsumerAdmin* or *NotifyConsumerAdmin* interface (with the exception of the *BasicLogFactory*) as shown in *Figure 4*.

There are five different log factory types; each type corresponds to the log type that it creates:

1. *BasicLogFactory* - simply creates BasicLogs: they cannot emit nor forward log events since they only inherit from the *LogMgr* interface.

1.2 Concepts and Architecture

2. *EventLogFactory* - creates *EventLogs* plus emits log events as untyped events. *EventLogFactory* inherits from both *LogMgr* and *CosEventChannelAdmin::ConsumerAdmin*.
3. *TypedEventLogFactory* - creates *TypedEventLogs* plus emits log events as untyped events. *TypedEventLogFactory* inherits from both *LogMgr* and *CosEventChannelAdmin::ConsumerAdmin*.
4. *NotifyLogFactory* - creates *NotifyLogs* plus emits log events as untyped events, which can be filtered. *NotifyLogFactory* inherits from both *LogMgr* and *CosNotifyChannelAdmin::ConsumerAdmin*.
5. *TypedNotifyLogFactory* - creates *TypedNotifyLogs* plus emits log events as untyped events, which can be filtered. *TypedNotifyLogFactory* inherits from both *LogMgr* and *CosNotifyChannelAdmin::ConsumerAdmin*.

The *LogMgr* interface is an abstract interface which provides the following operations common to all log factory interfaces:

- listing logs that have been created or copied
- listing logs by their id
- looking up logs by their id and returning a reference to the log

The *LogMgr* also provides log factories with log collection manager functionality.

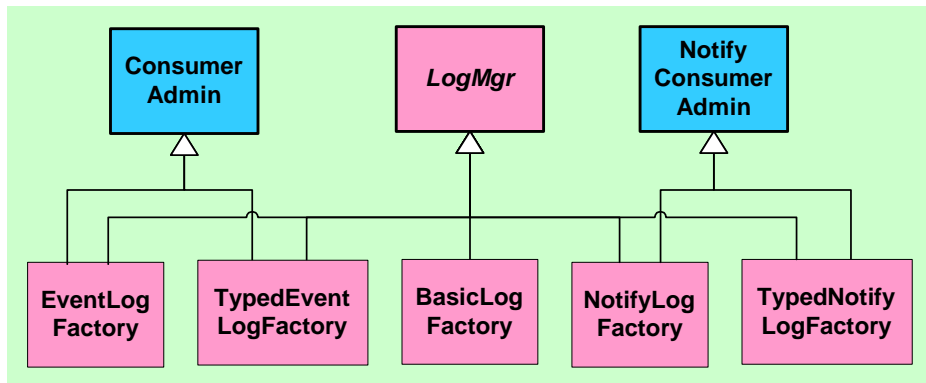


Figure 4 Log Factory Types and Inheritance

Log factories which inherit from the *ConsumerAdmin* or *NotifyConsumerAdmin* interfaces enable clients to receive *log generated events*. Log generated events contain information about the state or

condition of a log. These events are transmitted by the factory that created the log. A log factory transmits log generated events via its Consumer Admin object for all of the logs it has created, as shown in *Figure 5* below.

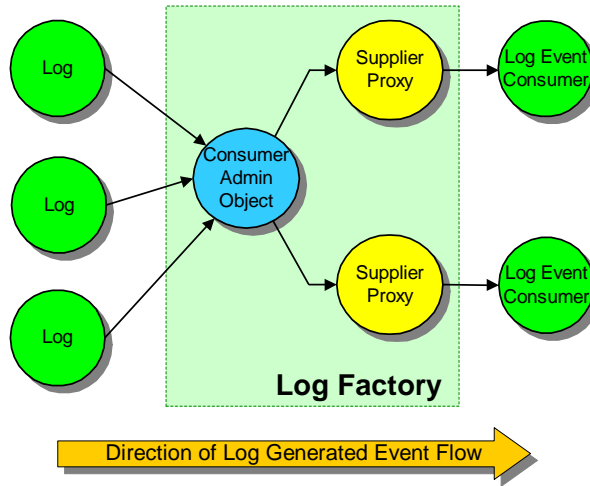


Figure 5 Log Generated Event Transmission

Log Networks

Logs can be formed into networks by connecting one log's consumer side (*ConsumerAdmin* interface) to the other logs' supplier side (*SupplierAdmin* interface), then forwarding events from each log to the others.

Components and Features

Log Records

Events or data sent to a log is stored as a log record in the log's persistent store. Events travel to the log's persistent store via the log's event channel; data sent from event-unaware clients is written directly to the persistent store.

There are two types of log record, *LogRecord* and *TypedLogRecord*. *LogRecord* is supported by all log interfaces. *TypedLogRecord* is only supported by the typed log interfaces. A typed log can be stored as a (non-typed) *LogRecord*, subject to the operations that a client uses to retrieve it.

A *LogRecord* consists of:

- *id* - a unique id number for the individual record
- *time* - a time stamp of when the event was logged (i.e. stored)
- *attr_list* (*attribute list*) - a user-defined list of attributes, which are not part of the event received by the log, containing log record related information; this information can be queried or modified
- *info* - the event or data stored as an *Any*, noting that structured or typed events can be wrapped in a CORBA *Any* before the event is stored

A *TypedLogRecord* consists of:

- *id* - a unique id number for the individual record
- *time* - a time stamp of when the event was logged (i.e. stored)
- *attr_list* (*attribute list*) - a user-defined list of attributes containing log record related information
- *interface_id* - repository id of the interface that sent the typed event
- *operation_name* - name of the operation that emitted the typed event
- *arg_list* - argument list that contains the event data

Features

Log records can be queried, retrieved, or deleted according to each record's id, log time, attributes, or event contents. Details of working with log records is described in Section 2, *Using Specific Features*.

Log Generated Events

Logs which inherit from an event channel are able to generate their own events (referred to as *log events*) which can be transmitted to consumer clients. Log generated events are not transmitted directly to consumers, but through the log factory that created the log. Log factories are also able to generate events.

Logs or log factories generate the following events:

- *ObjectCreation* - when a log is created
- *ObjectDeletion* - when a log is deleted
- *ThresholdAlarm* - the log's persistent store approaches its defined capacity or *wrapping* condition (wrapping is when the oldest records in the log are deleted in order to free space for new records)

- *AttributeValueChange* - when certain log attributes change (listed in *Table 1* on page 17)
- *StateChange* - a log's administrative or operational state changes (see *Log Management* below)
- *ProcessingErrorAlarm* - generated by the log factory when one of its logs generates an error

Consumers can receive log generated events by subscribing to the log factory.

Lifecycle Operations

Log supports the *copy* and *destroy* Lifecycle operations.

There are two versions of the copy operation. The first, *copy*, creates an empty log, generates an id, and initializes the new logs attributes to the same values as the log on which it was invoked. The second version, *copy_with_id*, likewise creates a empty log with its attributes initialized to the same values as the log on which it was invoked, but gives the log an id supplied as a parameter - provided the id does not already exist.

The *destroy()* method disconnects any consumers or suppliers connected to the log's event or notify channel, then destroys the logs and its log records. `destroy()` is inherited from `CosEventChannelAdmin::EventChannel`.



However, if an event or notification-based client terminates, then *it* must disconnect itself from the log (by destroying the relevant proxy object), since the log 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.



This need to remove proxy objects when a client terminates also applies to filter and other, similar objects - if a client or application creates an object, then it is responsible for removing the object, otherwise these “hanging” or unattached objects will remain in the system and will be a source of memory leaks.

Log Management

Logs have management operations for monitoring and setting their behaviour. Most of these operations are common to all logs; the event- and notify-based logs have additional operations for management of their particular features (listed under *Additional Management Operations* on page 16).

Several of the management operations cause *AttributeValueChange* events to be generated. The operations or changes which generate *AttributeValueChange* events are listed in *Table 1* on page 17.

Common Management Operations

The management capabilities which are common to all logs fall into two groups: those which can only obtain them and those which can change and obtain settings, values, and attributes.

Read-Only Settings

The following log settings, values, and attributes can only be obtained or read:

- *id* - the log's id
- *factory id* - the id of the log factory that created the log
- *operational state* - whether a log is operational or not (due to a run time problem, for example)

Read and Write Settings

The following log settings, values and attributes can be set as well as read:

- *administrative state* - turns logging on or off and termed *unlocked* and *locked*
 - when *unlocked*, log records are allowed to be created, retrieved, and deleted subject to the values of other states or settings
 - when *locked*, log records are allowed to be retrieved, and deleted, but new records are not allowed to be created
 - the administrative state does not affect the forwarding of events
- *maximum log size* - the maximum size, in bytes, that a log is allowed to store
 - a value of zero (0) sets the size as unlimited

- *log full actions* - what actions should be performed when the maximum log size (i.e. the size of its persistent store) has been reached. Possible actions are:
 - *wrap*, whereby the oldest records in the log are deleted in order to free space for new records
 - *halt*, whereby no more records will be logged and all incoming events are discarded
- *log capacity alarm threshold* - the level of the log's storage capacity at which an alarm is triggered to warn clients
 - the alarm threshold is set as a percentage of the maximum log size
 - the alarm threshold can be used to warn clients that the oldest records may be overwritten soon when the log full action is set to wrap; the alarm is triggered each time the log wraps
 - when the log full action is set to halt, the alarm can warn clients that the log will soon stop accepting events, provided
 - a) the client is registered to receive a ThresholdAlarm event (with the log's factory) and
 - b) the alarm threshold has been set to a value lower than 100% of the max log size
- *log duration* - the period of time period during which a log will store events, and where
 - the administrative and operational states are, respectively, unlocked and enabled
 - if the duration's *stop* value is set to zero (0), then the duration is indefinite
 - the duration is sufficiently long to avoid a race condition occurring between the activation of the log and the receiving of events
 - the default log duration is for the lifetime of the log (i.e. from the time the log is created until it is destroyed)
 - log duration does not affect the status of event forwarding
- *log scheduling* - the fine-grained time intervals that a log will store records of events, and where
 - the administrative and operational states are, respectively, unlocked and enabled

1.2 Concepts and Architecture

- time intervals can be specified according to minutes, hours, days, and weeks
- the time intervals must be within the log duration time interval
- the default is disabled, which means that the log can store records at any time
- log scheduling does not affect the status of event forwarding
- *availability status* - whether the combination of various states and conditions allows or enables events to be logged and/or retrieved; a log is available for logging if all of the following states are true:
 - the log is not full (i.e. the capacity of log store has not been reached)
 - the administrative state is unlocked
 - the operational state is enabled
 - the current time is within the log duration time
 - the current time is within the log's scheduled time(s)
- *log record lifetime* - the maximum length of time, in seconds, that a log record is retained
 - this is used to remove old, stale records from the log's store
 - the default lifetime value is zero (0), which allows records to remain indefinitely and until the log is destroyed
- *quality of service* - Quality of Service properties specific to logs (see *Quality of Service* on page 17); logs can also use the Notification Service's QoS properties provided by a log's inherited event or notify channel

Additional Management Operations

Logs which inherit from the event or notify channels provide additional management capabilities.

EventLog and TypedEventLog provide management of log forwarding.

NotifyLog and TypedNotifyLog provide management of log forwarding plus filtering of events

- received from suppliers and forwarded to consumers
- sent to the log record stores

These operations (for reading and setting) are briefly described below:

- *log record filtering* - sets the filter to be used for filtering log records (as distinct from the filters used by the log's channel for filtering incoming and forwarded events)
 - logs do not have a log record filter when they are created and therefore log all events (until a log record filter is set)
- *forwarding state* - controls whether a log will forward incoming events from all suppliers to all consumers which are currently connected
 - the default is set to *on* (forward all events)

Attribute Change Events

The following table lists properties or settings which cause *AttributeValueChange* events to be generated.

Table 1 Settings Generating AttributeValueChange Events

Setting or Value	Condition Generating Event
log full action	when set
log size	when set
log duration	when set
log scheduling	when <i>week mask</i> is set
max record lifetime	when set
log-specific quality of service	when log QoS set (excludes QoS inherited from event or notify channels)
log capacity alarm threshold	when set
forwarding state	when set
log filter	when set

Quality of Service

The Log Service provides Quality of Service (QoS) properties specifically for logs. In addition, logs which inherit from the Notify or TypedNotify channels can use the Notification Service's QoS framework (refer to the *OMG Notification Service Specification* or the *OpenFusion Notification Service Guide*).

The OMG specifies three `Log` specific QoS levels:

- *QoSNone* - no quality of service is promised
- *QoSFlush* - log records are made persistent



- the `QoSFlush` property does not affect the persistence setting for events which are *forwarded* by the log

- *QoSReliability* - all log records are guaranteed to be available



The OpenFusion implementation of the Log Service provides *QoSReliability* as the default (and only) `Log`-specific QoS level: this level guarantees that all log records sent to a log will be available and enables log records to be recovered if, for example, the system crashes.

Also, the OpenFusion `NotifyLog` inherits from the OpenFusion Notification Service, including all of its standard OMG and OpenFusion extended QoS properties.

Log Features Comparison

The following table provides a summary of the different features for each log type.

Table 2 Features of Different Log Types

Log Type	write (in-bound)	store (log record)	forward (out-bound)	emit log events	QoS
BasicLog	write operation	no filtering	no	no	log QoS
EventLog	untyped events push-pull model no filtering	no filtering	untyped events push-pull model no filtering	yes no filtering	log QoS
TypedEvent Log	typed events push-pull model no filtering	no filtering	typed events push-pull model no filtering	yes no filtering	log QoS
NotifyLog	untyped events structured events. push-pull model event filtering	log filter	untyped events structured events. push-pull model filtering	yes event filtering	log QoS Notification QoS
TypedNotify Log	typed events push-pull model event filtering	log filter	typed events push-pull model filtering	yes event filtering	log QoS Notification QoS

2 Using Specific Features

2.1 Introduction

The main tasks which are normally performed when using logs include:

- creation of the log for storing records of data or events sent by clients
- creation of clients which supply the data or events
- creation of clients which use the log's records

It may also be necessary to perform other tasks, such as:

- creating a client which can receive events sent via the log
- creating a client which can receive events generated by the log, itself
- enabling the log to record only selected events or only for specific time periods
- manage the log, in terms of its reliability, size, lifetime, etc.
- perform necessary clean-up operations

These tasks depend on the particular capabilities and implementation requirements of the type of log used (either *BasicLog* or *NotifyLog*).

This section, *Using Specific Features*, describes how the specific features and requirements for the *BasicLog* and *NotifyLog* types can be used to achieve the relevant tasks listed above. The section is organised into a sequence of topics which

- describes basic aspects common to both log types
- highlights aspects which are specific to the simple *BasicLog*, then progressing to Notification-based logs
- progressively introduces methods of performing specific tasks

Each topic uses examples to illustrate how tasks can be achieved. 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 Log and Naming Services are installed and running.

- There is little or no error-checking provided in the examples for the sake of clarity. Code to deal with exceptions has generally been omitted, again, for clarity and brevity, appreciating that exceptions must be properly caught and handled in a working system.
- Full details of the OpenFusion Log Service API is provided in the OpenFusion IDL Documentation.

Import statements

The following packages are required to be imported into classes which use the log types listed below.



This list is not exhaustive: additional packages may also be required, depending on requirements and the specific features of each service which are used.

BasicLog

Classes which create the log or clients which access the log

```
org.omg.DsLogAdmin.*
```

NotifyLog

Classes which create the log

```
org.omg.DsLogAdmin.*  
org.omg.DsNotifyLogAdmin.*  
org.omg.CosNotification.*
```

Clients which access the log

```
org.omg.DsLogAdmin.*  
org.omg.DsNotifyLogAdmin.*  
org.omg.CosNotification.*  
org.omg.CosNotifyComm.*  
org.omg.CosNotifyChannelAdmin.*
```

OpenFusion Extensions

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

```
com.prismt.cos.CosLogging.LogExtensions.*
```

2.2 The Basics and the BasicLog

This topic explains how to:

1. Create a BasicLog and perform rudimentary configuration of its record storage availability and behaviour.
2. Use the Naming Service to easily locate specific log instances by clients.
3. Create an event-unaware client which supplies the log with data.
4. Create a client which can query log records containing specific, desired information.

The `BasicLog` possesses features which are common to all logs. This log type is also relatively simple and straight-forward to use, since it inherits only from the Log interface. Consequently, it is a convenient log type to use for learning about the basic aspects of all log types.

An example application is employed here to demonstrate how to use a BasicLog. The application consists of two component programs: one creates a log, the other sends data to the log, then queries the log's records.



Evaluation copies of both the OpenFusion Log Service and OpenFusion Naming Service can be obtained from PrismTech's web site,

Creating a BasicLog

The first example program creates, configures and instantiates a BasicLog as a CORBA object. The program also uses the Naming Service to associate or *bind* the log object to a *name object*: this name object is subsequently used by the client program to obtain a reference (i.e. IOR) to the log object, thereby enabling the client to communicate with the log.

Import Statements

The example BasicLog creation program uses the services and associated APIs listed below. Applications or classes that use these services or APIs must include the import statements shown below.

OpenFusion Orb Adaptor

```
com.prismt.orb.ORBAdapter
```

BasicLog

```
org.omg.DsLogAdmin.*
```

Naming Service

```
org.omg.CosNaming.*  
org.omg.CosNaming.NamingContextPackage.*  
org.omg.CosNaming.NamingContextExt.*  
org.omg.CosNaming.NamingContextExtPackage.*
```

Initialisation

The program declares `orb` and `log` objects, plus an `org.omg.CORBA.IntHolder`, `logId`. The `logId` is used to hold the log's id (generated by the log factory which creates the log).

```
org.omg.CORBA.ORB orb = null;  
BasicLog log = null;  
org.omg.CORBA.IntHolder logId = null;
```

The OpenFusion's ORB-vendor independent abstraction layer is used to initialise and manage the orb object: the `ORBAdapter.init` method initialises the ORB:

```
orb = ORBAdapter.init (args);  
System.out.println ("got orb");
```

After the `orb` object has been initialised, the steps required for creating the BasicLog object can be performed.

Log Creation and Configuration

A log is created by a log factory's `create()` method. The `create()` method requires some configuration information, including the record store's maximum size and what happens when the store is full (i.e. overwrite the oldest records or reject new ones).

Any configuration properties or settings not explicitly set when the log is created are set to default values. All of a log's configuration properties or settings can be changed after the log is created using various Log methods, such as `set_interval()`. It is usually necessary to change one or more configuration properties, since many of the default settings disable potentially useful or required control mechanisms.

The following steps describe how to create and configure a BasicLog object.

Step 1: Obtain a BasicLog factory.

The following example code obtains a CORBA object for a *BasicLogFactory* using the ORB's *resolve_initial_references()* method, then narrows the object to a *BasicLogFactory*. Code should be included to test if the factory was not created, for example when the Log Service is not running.

```
// Create a BasicLog using the BasicLogFactory
org.omg.CORBA.Object object =
    orb.resolve_initial_references ("BasicLogFactory");
System.out.println ("initial references BasicLog factory");
BasicLogFactory factory = BasicLogFactoryHelper.narrow (object);
System.out.println ("resolved BasicLog factory.");
if(factory == null)
{
    System.out.println ("BasicLog factory not found.");
    System.exit(1);
}
```

Step 2: Create the log.

A *BasicLog* is created with the *BasicLogFactory.create()* method. In this example, the log, called *log*, will be created with a record store having a maximum size of 10,000 bytes and which will overwrite the oldest records when full (set using *wrap.value*). The record store size used here is for illustrative purposes only: the size used in a real environment should reflect the approximate size (in bytes) of the records to be stored multiplied by the number of records the log is required to hold.

The *logId* variable (an *IntHolder*) is used to hold the log's id value (assigned by the *create()* method).

```
// create a log which will remove oldest records when its
// size of 10,000 bytes is exceeded
logId = new org.omg.CORBA.IntHolder();
log = factory.create(wrap.value, 10000, logId);
```

Step 3: Configure the log.

Administrative or other property settings can now be set, as required. It is recommended that the log's ability to record data or events be disabled whenever an administrative or similar property is set using *set_administrative_state()* with *AdministrativeState.locked*.

The example uses the log's *set_interval()* method to set the length of time that the log will be allowed to store records. This method takes a *TimeInterval* parameter; *TimeInterval* has two values, *start* and *stop*, which define the beginning and end times of the interval. *TimeInterval* is based on *Gregorian* time: any time value used should be relative to the start

of the Gregorian calendar, i.e. the 15th of October 1582. This time can easily be obtained using Java's *GregorianCalendar* and *Date* classes as shown in the example.

When all necessary properties have been set, then re-enable record storing using `set_administrative_state()` with `AdministrativeState.unlocked`.

```
// lock the log prior to setting its operational interval
log.set_administrative_state(AdministrativeState.locked);

// Set length of time for log to accept data
GregorianCalendar calendar = new GregorianCalendar();
Date beginning = calendar.getGregorianChange();
long basetime = - beginning.getTime();
long now = System.currentTimeMillis();

TimeInterval interval = new TimeInterval();
interval.start = now + basetime;
interval.start *= 10000; // convert to 100ns

// Stop logging from now plus duration
interval.stop = now + basetime + duration;
interval.stop *= 10000; // convert to 100ns
log.set_interval(interval);

// unlock the log after setting the operational interval
log.set_administrative_state(AdministrativeState.unlocked);
```

Locating the Log

Any client or application which uses a CORBA object must be able to obtain a reference to it using its IOR. One method is to save its IOR to a file, then have the client read the file in order to obtain the IOR. However, from the viewpoint of a distributed system, this method is very crude and has obvious failings, such as “How does the client locate the file containing the IOR?”.

An alternate, much preferred method is to use the Naming Service to locate the object on behalf of the client. The Naming Service associates objects with a name: this association is called a *name binding* and can be used by clients to efficiently obtain a reference to the object. The only prerequisites for using name bindings are that the Naming Service must be running and the object must be bound to a name before clients try to use it.

This example binds the log object to a name using the Naming Service: clients can then use the name to locate the log. Describing how to use the Naming Service and create name bindings is outside the scope of this topic. For convenience however, a brief description is provided in Appendix A, *Locating a Log with the Naming Service*, on page 101. Refer to the *OpenFusion Naming Service Guide* for complete details on using the Naming Service.

```
// bind the log object to a name with Naming Service
org.omg.CORBA.Object namingService = null;
```

```

NamingContextExt rootContext = null;
NameComponent basicLog[];

namingService = orb.resolve_initial_references ("NameService");
rootContext = NamingContextExtHelper.narrow(namingService);
basicLog = new NameComponent [1];

basicLog[0] = new NameComponent ("BasicLog", "log");
rootContext.rebind(basicLog, log);

System.out.println("BasicLog created.");

```

The log object can now be located, from clients located anywhere in the system, using the context's *resolve()* method (described below under *Creating a Client for the BasicLog* on page 25).

BasicLog Exceptions

The example code shown above, which creates the BasicLog object and Naming Service name binding, must either throw or catch the exceptions listed below. Log exception details, for this and all other examples, are described in Section 3.1, *Exceptions*, on page 73.

ORB

org.omg.CORBA.ORBPackage.InvalidName

BasicLog (including Interval)

org.omg.DsLogAdmin.InvalidLogFullAction

org.omg.DsLogAdmin.InvalidTime

org.omg.DsLogAdmin.InvalidTimeInterval

Naming Service

org.omg.CosNaming.NamingContextPackage.InvalidName

org.omg.CosNaming.NamingContextPackage.NotFound

org.omg.CosNaming.NamingContextPackage.CannotProceed

Creating a Client for the BasicLog

The following client program repeatedly sends data to a BasicLog instance and then queries the log's records to selectively obtain the records of data previously sent. The client utilises separate threads, via J2SE's *Timer* and *TimerTask* classes, to send data and query the log.

Import Statements

The program uses the services and associated APIs listed below. Applications or classes that use these services or APIs must include the import statements shown below.

OpenFusion Orb Adaptor

```
com.prismt.orb.ORBAdapter
```

BasicLog

```
org.omg.DsLogAdmin.*
```

Naming Service

```
org.omg.CosNaming.*  
org.omg.CosNaming.NamingContextPackage.*  
org.omg.CosNaming.NamingContextExt.*  
org.omg.CosNaming.NamingContextExtPackage.*
```

J2SE'S Timer and TimerTask

```
java.util.*  
java.util.TimerTask
```

Client Initialisation

The program declares `orb` and `log` objects, plus an `org.omg.CORBA.Any` array, `data`, which will contain the data that will be sent to the log. Since the Naming Service will be used to retrieve the `log` object, variables required to retrieve the Naming Service instance's `context` and `name component` for the log are also declared.

As mentioned above, this example uses threads provided by the `Timer` and `TimerTask` classes. Two `Timer` objects, `sendTimer` and `receiveTimer`, are declared. These thread objects respectively send data and queries at set intervals for a fixed duration of time, as defined by the `interval` and `duration` variables. The `delay` variable sets a time delay, in milliseconds, for the commencement of the query thread. Please note that `Timer` and `TimerTask` are not the only approaches which can be used to send data or events to a log using threads.

```
org.omg.CORBA.ORB orb = null;  
org.omg.CORBA.Object obj = null;  
  
BasicLog log = null;  
org.omg.CORBA.Any[] data;  
  
NamingContextExt rootContext = null;  
NameComponent basicLog[];
```

```
private Timer sendTimer;
private Timer receiveTimer;

private final long duration = 3 * 60 * 1000; // activity time in millisecs
private final long interval = 2000; // 2 seconds
private final long delay = 2000;
```

The example client obtains references to and performs initialisation of the ORB, Naming Service instance, naming context, and name component in a similar manner as was performed by the previous program which created the log.

A reference to the log is obtained by passing a `NameComponent` object (containing the log's `id` and `kind` values) to the context's `resolve()` method, then narrowing the object to the `BasicLog` type (`resolve()` returns an *Object* type).

```
// obtain reference to the orb
orb = ORBAdapter.init(args);

// obtain reference to log using Naming Service
obj = orb.resolve_initial_references("NameService");
rootContext = NamingContextExtHelper.narrow(obj);
basicLog = new NameComponent[1];
basicLog[0] = new NameComponent("BasicLog", "log");
obj = rootContext.resolve(basicLog);
log = BasicLogHelper.narrow(obj);
```

Preparing the Data

The `BasicLog` type can not accept events - it can only accept data which is sent in an array of type `Any`. The following code creates an `Any` array and initialises it with four different primitive types: `string`, `double`, `long` and `char`. If a single data element is to be sent, then an `Any` array with one element should be used.

```
// initialise sample data to be logged
data = new org.omg.CORBA.Any[4];

for(int i = 0; i < data.length; i++)
{
    data[i] = orb.create_any();
}
data[0].insert_string("Hello world!");
data[1].insert_double(42.424242);
data[2].insert_long(123456789);
data[3].insert_char('a');
```

Sending the Data

The example code shown below:

- checks that a log is available to accept records
- writes data to the log

The client sends data to the log every two seconds for five minutes: the frequency and duration is controlled by the `Timer.schedule()` method in combination with the client's implementation of `TimerTask.run()`. The `run()` method is executed by `schedule()` according to the `delay` and `interval` values. `run()` has been implemented so that it exits when a timeout value, `stopTime`, is reached.



It should be remembered that the use of `Timer` and `TimerTask.run()` is a client specific approach to controlling *when* the log is accessed. (Details for using `Timer` and `TimerTask` are provided in Sun Microsystems's J2SE API documentation and the *Java Tutorial*, available on Sun's web site).

The following code is taken from the client's `TimerTask.run()` implementation.

```
// thread for sending data
class Sender extends TimerTask
{
    long stopTime = System.currentTimeMillis() + duration;

    public void run()
    {
        if((System.currentTimeMillis() < stopTime) &&
            (log.get_availability_status().off_duty == false))
        {
            try
            {
                log.write_records(data);
                System.out.println ("\nWrote " + data.length + " records");
            }
            catch (org.omg.DsLogAdmin.LogFull ex)
            {
                System.err.println("Log is full.");
                System.exit(1);
            }
            catch (org.omg.DsLogAdmin.LogLocked ex)
            {
                System.err.println("Log is locked.");
                System.exit(1);
            }
            catch (org.omg.DsLogAdmin.LogDisabled ex)
            {
                System.err.println("Log is disabled.");
                System.exit(1);
            }
            catch (org.omg.DsLogAdmin.LogOffDuty ex)
            {
                System.err.println("Log is off duty.");
                System.exit(1);
            }
        }
        else
        {
            if(log.get_availability_status().off_duty)
            {
                System.out.println("The log is off duty " +
                    "- cannot log data.");
            }
            if(System.currentTimeMillis() >= stopTime)
            {
                System.out.println("Finished sending data.\n");
                sendTimer.cancel();
            }
        }
    }
}
```

```

    }
  } // end run
} // end inner class Sender

```

The `get_availability_status().off_duty` value is used to determine if the log is available to store records: the client uses this method to test to whether it can write data to the log. A log can become *off duty*, i.e. unavailable to receive data or events, for several reasons, including:

- the log's operational state is disabled
- the log's administrative state is locked (refer to the code showing `log.set_administrative_state(AdministrativeState.locked)` under *Log Creation and Configuration* above)
- the current time is outside the log's *duration time* (refer to the code showing `log.set_interval(interval)` under *Log Creation and Configuration* above)
- the current time is outside the log's scheduled times (scheduled times are set using the Log's `set_week_mask()` method)

Provided the log is available, the data (as an array of `Any`s) is sent to the log by simply using:

```
log.write_records(data);
```

Each `Any` in the array will be stored as a separate record; if the array contains four elements, then four records will be stored.

Querying the Log Records

The code from the example client shown below:

- creates a simple query which retrieves selected log records
- retrieves information from the record

The client begins querying the log's records two minutes after it starts sending data, every two seconds for eight minutes: the frequency and duration is controlled using the same approach that is used for sending the data, i.e. using `Timer.schedule()` and a `TimerTask.run()` implementation as explained under *Sending the Data* above.

A query consists of two elements:

- a constraint expression, stated according to a supported grammar and constraint language, e.g. `EXTENDED_TCL`, and

- making the query using Log's `query()` method with the constraint expression, grammar, and iterator parameters (`query()` sets the iterator via an iterator holder)

The query will return a `LogRecord` array of those records which satisfy the conditions defined in the constraint.

A `LogRecord` is a structure containing the following elements:

- `id` - a unique number which is assigned by the log
- `time` - the time the log was created (based on the Gregorian calendar, expressed in 100 nanosecond units (IDL type `TimeT`))
- `attr_list` - an (optional) attribute list of user defined name-value pairs which can provide information about the date or event, but which is not part of the data, as such
- `info` - the actual data or event being stored

Step 1: Creating a Query Expression

A `LogRecord` array is created to hold records which are returned by the query. The default grammar (the OpenFusion Log Service uses `EXTENDED_TCL`) is assigned to a string which will be passed to `query()`. The constraint expression states, in this case, that only records that contain the string "Hello World!" should be retrieved. Note the single quotes surrounding the 'Hello world!' data string. The `query()` method returns the number of query results (via `IteratorHolder`).

```
// thread for querying log records
class Receiver extends TimerTask
{
    // initialise sample query
    LogRecord[] records;
    String grammar = default_grammar.value;
    String constraint = "$.info == 'Hello world!'";
    IteratorHolder iter = new IteratorHolder();

    long stopTime = System.currentTimeMillis() + duration;
```

Step 2: Sending the Query

The following code executes the query, saves the result to the `LogRecord` array, called `records`, then retrieves and prints the id number and the data which was stored in the last record stored in `records`.

The query is called within a `try-catch` block in order to catch the `InvalidGrammar` and `InvalidConstraint` exceptions.

```
public void run()
{
```



```

if(System.currentTimeMillis() < stopTime)
{
    try
    {
        records = log.query (grammar, constraint, iter);
        if (records.length >= 1)
        {
            int last = records.length - 1;
            System.out.println ("The last log record\'s id is: " +
                records[last].id + " and contains " +
                records[last].info);
        }
        else
        {
            System.out.println ("No records in log store.\n");
        }
    }
    catch (org.omg.DsLogAdmin.InvalidGrammar ex)
    {
        System.err.println ("Invalid grammar.");
        System.exit (1);
    }
    catch (org.omg.DsLogAdmin.InvalidConstraint ex)
    {
        System.err.println ("Invalid constraint.");
        System.exit (1);
    }
}
else
{
    System.out.println("Finished querying log records.\n");
    receiveTimer.cancel();
}
} //end inner class Receiver

```

Client Exceptions for the BasicLog

The code used in the example client program must throw or catch the following exceptions.

ORB

org.omg.CORBA.ORBPackage.InvalidName

BasicLog (including Write and Query Related)

org.omg.DsLogAdmin.LogDisabled
 org.omg.DsLogAdmin.LogOffDuty
 org.omg.DsLogAdmin.LogFull
 org.omg.DsLogAdmin.LogLocked
 org.omg.DsLogAdmin.InvalidGrammar
 org.omg.DsLogAdmin.InvalidConstraint

Naming Service

org.omg.CosNaming.NamingContextPackage.NotFound
org.omg.CosNaming.NamingContextPackage.CannotProceed
org.omg.CosNaming.NamingContextPackage.InvalidName

2.3 The NotifyLog and Event-style Events

This topic describes the essential aspects for working with the NotifyLog. Understanding these aspects is a precursor to understanding the NotifyLog's more powerful features. The topic explains how to:

1. Create an NotifyLog and perform initial configuration for this log type.
2. Set log scheduling using the `set_week_mask()` method.
3. Create a client which supplies the log with Event-style events.
4. Create a client which receives Event-style events forwarded by the log and also retrieves the latest log records.
5. Perform clean-up operations when terminating NotifyLog clients.

The NotifyLog inherits from both the `Log` interface and the Notification Service's event channel interfaces. (see Figure 1, *Log Types and Inheritance*, on page 7) Consequently, it is more complex than the BasicLog and requires an understanding of the Notification Service's proxies and channels. Developing an understanding of proxies and channels is important generally since they are used by all log types except the BasicLog.

An example application consisting of three program components is used here to demonstrate how to create and interact with a NotifyLog. The three component model used by the example application reflects an architecture which may likely be used in a working environment.

Creating a NotifyLog

This example program creates, configures and instantiates a NotifyLog as a CORBA object. The program also uses the Naming Service to bind the log object to a name, as was used in the BasicLog example program, to facilitate location of the log by the application's client programs.

Import Statements

This example NotifyLog creation program uses the services and associated APIs listed below. Applications or classes that use these services or APIs must include the import statements shown below.

OpenFusion Orb Adaptor

```
com.prismt.orb.ORBAdapter
```

NotifyLog

```
org.omg.DsLogAdmin.*
```

```
org.omg.DsNotifyLogAdmin.*
org.omg.CosNotification.*
```

Naming Service

```
org.omg.CosNaming.*
org.omg.CosNaming.NamingContextPackage.*
org.omg.CosNaming.NamingContextExt.*
org.omg.CosNaming.NamingContextExtPackage.*
```

NotifyLog Initialisation

The orb object for the program is declared and initialised in the same manner as was shown under Section 2.2, *The Basics and the BasicLog*, on page 21.

NotifyLog Creation and Configuration

A NotifyLog is created by obtaining a reference to a *NotifyLogFactory*, then using the factory to create a *NotifyLog* object with the requisite configuration information. The *NotifyLogFactory*'s `create_with_id()` method requires:

- the record store's maximum size, in bytes
- whether to wrap (overwrite the oldest records) or reject new records when the record store is full (*halt.value*)
- the levels when alarms are generated as the record store becomes full, expressed as a percentage of the record store's maximum size
- the initial Quality of Service (QoS) property settings
- the initial Admin property settings

It is possible to avoid setting the QoS or Admin properties by passing zero length *Property* arrays to the create methods.

Any remaining configuration properties which are required to be set to non-default values can be set after the log is created.



The OMG specifies three Log specific QoS levels: *QoSNone*, *QoSFlush*, and *QoSReliability*. The OpenFusion implementation of the Log Service provides *QoSReliability* as the default (and only) Log-specific QoS level: this level guarantees that all log records sent to a log will be available and enables log records to be recovered if, for example, the system crashes.

Also, the OpenFusion NotifyLog inherits from the OpenFusion Notification Service, including all of its standard OMG and OpenFusion extended QoS properties.

Step 1: Obtain a NotifyLog factory.

The following example code obtains a CORBA object for a *NotifyLogFactory* using the ORB's *resolve_initial_references()* method, then narrows the object to a *NotifyLogFactory*. Code should be included to test whether the factory was successfully created.

```
orb = ORBAdapter.init(args);
org.omg.CORBA.Object object =
    orb.resolve_initial_references("NotifyLogFactory");
NotifyLogFactory factory = NotifyLogFactoryHelper.narrow(object);

if(factory == null)
{
    System.out.println("Can not resolve initial reference " +
        "of NotifyLogFactory. ");
    System.exit(1);
}
```

Step 2: Create the log.

The NotifyLog is created with *EventLogFactory.create_with_id()*. In this example, the NotifyLog, called *log*, will:

- have a record store having a maximum size of 10,000 bytes¹
- overwrite the oldest records when full (using *wrap.value*)
- generate alarms when the log is 25, 50, and 75 percent full (defined with an array of shorts called *thresholds*)

The *logId* variable (an *IntHolder*) is used to hold the log's id value (which is assigned by *create_with_id()*)

```
NotifyLog log = null;
int logId = 0;
short[] thresholds = {25, 50, 75};

Property[] qos = new Property[0];
Property[] admin = new Property[0];

// create new log, replacing existing log having same id if it exists
org.omg.CORBA.Object corbaObj = null;
corbaObj = factory.find_log(logId);
if ( corbaObj != null )
{
```

-
1. The size used in a real environment should reflect the approximate size (in bytes) of each record to be stored multiplied by the number of records the log is required to hold.

```

        System.out.println("Existing log found and being replaced.");
        log = NotifyLogHelper.narrow(corbaObj);
        log.destroy();
    }

    log = factory.create_with_id(logId, wrap.value, 10000,
        thresholds, qos, admin);

```

Note that `factory.find_log(logId)` is used to determine if a log with the same id already exists - if does, then it is destroyed (with `destroy()`).

i

`destroy()` removes the log, with its log store, and disconnects any clients which are connected to the log: it should be used to remove the log, but with caution, since the careless use of `destroy()` could result with the unintentional loss of records.

Step 3: Configure the log.

The process for configuring a NotifyLog is similar to the process shown above for the BasicLog. This example shows how to configure the intervals that a log is allowed to store records using `WeekMask`. `WeekMask` is more flexible than `TimeInterval` (used in the BasicLog example), such that multiple start and stop times can be set to occur on specified days of the week. (`WeekMask` uses 24 hour time in the range of 0-23 hours and 0-59 minutes, as opposed to the Gregorian time used by `TimeInterval`.)

The following code creates a schedule that allows the log to store records on Mondays, Tuesdays, Wednesdays, Thursdays, and Fridays, for two intervals, from 0900 to 1200 and from 1300 to 1700, each day. After the day and time values are set, `log.set_week_mask(masks)` is called to set the schedule.

Note that the log's administrative state is locked while the `WeekMask` is being set. This is done in order to stop records being stored while the properties are being changed.

```

// lock the log prior to setting its operational interval
log.set_administrative_state(AdministrativeState.locked);

// Set length of days and times for log to accept data
WeekMaskItem[] masks = new WeekMaskItem[1];
masks[0] = new WeekMaskItem();
masks[0].days = (short) (Monday.value | Tuesday.value | Wednesday.value
    | Thursday.value | Friday.value);
masks[0].intervals = new Time24Interval[2];

masks[0].intervals[0] = new Time24Interval();
masks[0].intervals[0].start = new Time24((short) 9, (short) 0);
masks[0].intervals[0].stop = new Time24((short) 12, (short) 0);

masks[0].intervals[1] = new Time24Interval();
masks[0].intervals[1].start = new Time24((short) 13, (short) 0);
masks[0].intervals[1].stop = new Time24((short) 17, (short) 30);

log.set_week_mask(masks);

```

```
// unlock the log after setting the operational interval
log.set_administrative_state(AdministrativeState.unlocked);
```

Locating the NotifyLog

This example, like the BasicLog example, uses the Naming Service to facilitate the locating of the log object by clients. The code used here is nearly identical to the equivalent code used for the BasicLog example, however the `id` value of the `NameComponent` has been changed to `"NotifyLogA"` in order to distinguish this log object from the BasicLog object used previously and the `NotifyLog` object to be used in the *NotifyLog, Structured Events and More* example on page 48.¹

```
// bind log object to a name using the Naming Service
org.omg.CORBA.Object namingService = null;
NamingContextExt rootContext = null;
NameComponent notifyLog[];

namingService = orb.resolve_initial_references ("NameService");
rootContext = NamingContextExtHelper.narrow(namingService);
notifyLog = new NameComponent[1];
notifyLog[0] = new NameComponent("NotifyLogA", "log");
rootContext.rebind(notifyLog, log);
System.out.println("NotifyLogA created");
```

NotifyLog Exceptions

The example code shown above must either throw or catch the exceptions listed below. Log exception details, for this and all other examples, are described in Section 3.1, *Exceptions*, on page 73.

ORB and General (connecting to services)

`org.omg.CORBA.ORBPackage.InvalidName`

NotifyLog and Duration

`org.omg.DsLogAdmin.LogIdAlreadyExists`
`org.omg.DsLogAdmin.InvalidLogFullAction`
`org.omg.DsLogAdmin.InvalidTime`
`org.omg.DsLogAdmin.InvalidTimeInterval`
`org.omg.DsLogAdmin.InvalidThreshold`
`org.omg.DsLogAdmin.InvalidMask`
`org.omg.CosNotification.UnsupportedQoS`
`org.omg.CosNotification.UnsupportedAdmin`

1. Although `"NotifyLogA"` is used here, any string value could be used, provide the `id` and `kind` combination is unique within the naming context.

Naming Service

```
org.omg.CosNaming.NamingContextPackage.InvalidName
org.omg.CosNaming.NamingContextPackage.NotFound
org.omg.CosNaming.NamingContextPackage.CannotProceed
```

Creating a Supplier Client for the NotifyLog

The following supplier client program repeatedly sends Event-style events to a NotifyLog instance. This example client utilises J2SE's *Timer* and *TimerTask* classes to control how long and at what interval to send the events.

The most important difference between this client and the client used for the BasicLog example is that this client connects to the log as an *event supplier* - that is, it supplies events to the NotifyLog's *event channel* via a *proxy* (remembering that the NotifyLog inherits, via the *NotifyChannel*, from the *EventChannel* - as well as from *Log*). The client for the BasicLog wrote data directly to the log store, whereas this client sends events to the NotifyLog's event channel; the log, itself, writes the events to the record store, as well as forwarding them to consumer clients.

Accordingly, the supplier client uses two *CosNotify* interfaces (see *Import Statements* below), plus it must implement *PushSupplierOperations*.

i

Classes which implement any of the *Operations* interfaces must use *com.prismt.orb.ObjectAdapter* methods. Accordingly, unlike the previous examples, this client uses *ObjectAdapter* instead of *ORBAdapter* to connect with and initialise the ORB. However, there may be situations in working applications where methods from *both* *ObjectAdapter* and *ORBAdapter* are needed.

The *PushSupplierOperations* methods which are required to be implemented by the supplier are *disconnect_push_supplier()* and *subscription_change()*. These are callback methods which enables the NotifyLog's event channel to respectively inform the client that it has been disconnected or that there has been a subscription change.

```
// PushSupplierOperations callback method
public void disconnect_push_supplier()
{
    System.out.println ("Disconnected by proxy");
} // end disconnect_push_supplier

public void subscription_change(org.omg.CosNotification.EventType[] added,
    org.omg.CosNotification.EventType[] removed)
{
    System.out.println ("Supplier subscription changed");
} // end subscription_change
```

Import Statements

The example supplier program uses the services and associated APIs listed below. Applications or classes that use these services or APIs must include the import statements shown below.

OpenFusion Orb Adaptor

```
com.prismt.orb.ObjectAdapter
```

NotifyLog

```
org.omg.DsLogAdmin.*  
org.omg.DsEventLogAdmin.*  
org.omg.CosEventComm.*  
org.omg.CosEventChannelAdmin.*
```

Naming Service

```
org.omg.CosNaming.*  
org.omg.CosNaming.NamingContextPackage.*  
org.omg.CosNaming.NamingContextExt.*  
org.omg.CosNaming.NamingContextExtPackage.*
```

J2SE'S Timer and TimerTask

```
java.util.*  
java.util.TimerTask
```

Supplier Client Initialisation

The program declares, as in the previous examples, *orb* and *log* objects, plus an *org.omg.CORBA.Any*, called *event*, which is the Event-style event that will be sent to the log. A *ProxyPushConsumer* object is also declared, since the program will connect to the consumer client via the log's push consumer proxy (see *The Client as an Event Supplier* on page 40 and *Creating a Consumer Client for the NotifyLog* on page 43).

Since the Naming Service will be used to retrieve the *log* object, associated variables required to locate the log via the Naming Service are also declared (refer to Section 2.2, *The Basics and the BasicLog*, on page 21 and Appendix A, *Locating a Log with the Naming Service*, on page 101 for details).

Like the previous examples, the *Timer* and *TimerTask* classes control the interval and duration when events are sent, as specified by the *interval* and *duration* variables.

```
// event suppliers must implement supplier operations  
public class NotifySupplierA implements PushSupplierOperations
```



```

{
    org.omg.CORBA.ORB orb = null;
    org.omg.CORBA.Object obj = null;

    NotifyLog log = null;
    org.omg.CORBA.Any event;
    ProxyPushConsumer consumerProxy = null;

    NamingContextExt rootContext = null;
    NameComponent notifyLog[];

    private Timer sendTimer;

    private final long duration = 3 * 60 * 1000; // 3 seconds in millisecs
    private final long interval = 2000; // 2 seconds
    private final long delay = 0;

```

The client program then

- initialises the ORB
- establishes the client as a transient CORBA object
- obtains a reference to the NotifyLog instance using the Naming Service
- establishes itself as a *PushSupplier* and
- obtains a reference to the log's *push consumer proxy*, thereby connecting itself to the log (described under *The Client as an Event Supplier* below)

When these tasks are completed, the client informs the ORB that it is ready to accept incoming requests (the only requests that might be received by this example will be from `disconnect_push_supplier()`).

```

orb = ObjectAdapter.init(args);

// instantiate this client as a transient CORBA object
ObjectAdapter.createTransient(this);

// obtain reference to log using Naming Service
obj = orb.resolve_initial_references("NameService");
rootContext = NamingContextExtHelper.narrow(obj);
notifyLog = new NameComponent[1];
notifyLog[0] = new NameComponent("NotifyLogA", "log");
log = NotifyLogHelper.narrow(rootContext.resolve(notifyLog));

// establish this object as a push supplier and
// connect to the log's proxy consumer
PushSupplier supplier =
    PushSupplierHelper.narrow(ObjectAdapter.getObject(this));
SupplierAdmin supplierAdmin = log.default_supplier_admin();
org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder();
ProxyConsumer proxy =
    supplierAdmin.obtain_notification_push_consumer(
        ClientType.ANY_EVENT, id);

consumerProxy = ProxyPushConsumerHelper.narrow(proxy);
consumerProxy.connect_any_push_supplier(supplier);

// enable incoming requests, without blocking, for log callbacks
ObjectAdapter.ready(false);

```

The Client as an Event Supplier

Any client which sends or receives events to or from an event channel must be narrowed to an (Event Service) supplier or consumer type. Further, suppliers and consumers will use one of two models to send or receive events: *push* or *pull*.

i The push model is the only model supported by the OpenFusion Log Service implementation since it model which is typically or generally used. Briefly:

- *push suppliers* actively push events to the event channel
- *push consumers* passively receive events sent by the channel, i.e. a push consumer will receive any event that is sent to it by the event channel
- *pull suppliers* have their events pulled from them by the channel, i.e. the supplier must supply an event (if it has one), whenever the channel makes a request
- *pull consumers* actively pull events from the event channel, i.e. they retrieve events whenever they want them

Our supplier client is a *PushSupplier*. Its consumer client, described below, is a *PushConsumer*.

Clients connect to an event channel through a *proxy*: the proxy represents the consumer or supplier object(s) that a client intends to transmit events to or receive events from: a *supplier* connects to a *consumer* proxy, and visa versa.

The proxy, in combination with an event channel, enables a client to connect transparently with any number of other clients, without the need to obtain a reference to each client that is added to the system.

i A detailed discussion of these issues is beyond the scope of this topic, however the OMG's *Event Service Specification* and *OpenFusion Notification Service Guide* provide a complete description of push and pull models, suppliers, consumers, proxies and related topics.

Connecting a Client to an Event Channel

A connection between a client and an event channel requires the following:

Step 1: Narrow the client to the appropriate type, for example:

```
PushSupplier supplier =  
    PushSupplierHelper.narrow(ObjectAdapter.getObject(this));
```

where *this*, in this example, is the client itself

Step 2: Obtain a reference to the channel's *admin* object for the respective supplier or consumer. Since the log is a subtype of a notification channel (which is a subtype of an event channel), call *NotifyLog.default_supplier_admin()* to obtain the channel's default supplier admin object:

```
SupplierAdmin supplierAdmin =
    log.default_supplier_admin();
```

Step 3: Obtain a reference from the admin object to the proxy (remembering that suppliers connect to consumer proxies and visa versa):

```
ProxyConsumer proxy =
    supplierAdmin.obtain_notification_push_consumer(
        ClientType.ANY_EVENT, id);

ProxyPushConsumer consumerProxy =
    ProxyPushConsumerHelper.narrow(proxy);
```

Step 4: Connect the client to the proxy:

```
consumerProxy.connect_any_push_supplier(supplier);
```

Step 5: Inform the ORB that the client is ready to supply events:

```
ObjectAdapter.ready(false);
```

Sending Events

The client sends a series of Event-style events, *AnyS*, to the log. Like the *BasicLog* example, the time interval and duration of when the events are sent is controlled by *Timer* and an implementation of the abstract *TimerTask* class' *run()* method.

The *TimerTask* implementation, *Sender.run()* shown below, uses *stopSendingTime* to define when to stop sending events; the *num* variable is used to provide a data value for each event and is incremented each time an event is sent.

```
// thread for sending data
class Sender extends TimerTask
{
    // data value to be placed in each event
    int num = 0;

    long stopSendingTime = System.currentTimeMillis() + duration;
```

Provided the current time is earlier than *stopSendingTime* and the log's *off_duty* property is false, then a *num* value will be inserted into each event. The event is then sent to the log via the log's consumer proxy, *consumerProxy.push(event)*.

An `org.omg.CosEventComm.Disconnected` exception is thrown if the client becomes disconnected from the proxy.

```
public void run()
{
    if((System.currentTimeMillis() < stopSendingTime) &&
        (log.get_availability_status().off_duty == false))
    {
        try
        {
            num++;
            event = orb.create_any();
            event.insert_long(num);
            consumerProxy.push(event);
            System.out.println ("Event sent with value " + num + ".");
        }
        catch (org.omg.CosEventComm.Disconnected ex)
        {
            System.exit(1);
        }
    }
    else
    {
        if(log.get_availability_status().off_duty)
        {
            System.out.println("The log is off duty " +
                "- cannot log event.");
        }
        if(System.currentTimeMillis() >= stopSendingTime)
        {
            System.out.println("Finished sending events.\n");
        }
    }
}
```

Terminating the Supplier Client

Before an event-based client terminates it must destroy any proxy objects that it has created. If these proxies are not destroyed, then they will become a source of leakage. Also, logs have no means of knowing if a client terminates unless the proxy is destroyed: it is therefore the client's responsibility to disconnect from a log by destroying the proxy. A client disconnects from event-based logs and destroys the associated proxy using an appropriate proxy disconnect method.



The OMG naming convention for disconnecting a proxy is the reverse of its equivalent connection method, whereby the *supplier* part of the method's name is replaced by *consumer* - which can be confusing. The easiest means of clarifying the convention is by example:

- a push supplier called *supplier connects* to a `ProxyPushConsumer` instance called `consumerProxy` using `connect_push_supplier(supplier)`
- a push supplier called *supplier disconnects* from a `ProxyPushConsumer` instance called `consumerProxy` using `disconnect_push_consumer()`

The example client disconnects from the proxy when `Sender.run()` has finished sending events.

```

        // cleanup - destroy proxy, stop incoming requests
        consumerProxy.disconnect_push_consumer();
        ObjectAdapter.shutdown();

        // stop thread
        sendTimer.cancel();
    }
} // end run
} // end inner class Sender

```

Supplier Client Exceptions

The example code shown above, which creates a NotifyLog supplier client and uses a Naming Service name binding, must either throw or catch the exceptions listed below.

ORB

```
org.omg.CORBA.ORBPackage.InvalidName
```

NotifyLog

```
org.omg.CosNotifyChannelAdmin.AdminLimitExceeded
org.omg.CosEventChannelAdmin.AlreadyConnected
```

Naming Service

```
org.omg.CosNaming.NamingContextPackage.InvalidName
org.omg.CosNaming.NamingContextPackage.NotFound
org.omg.CosNaming.NamingContextPackage.CannotProceed
```

Creating a Consumer Client for the NotifyLog

The example consumer client program:

- passively receives events, which have been forwarded by the log from the supplier
- retrieves the most recent ten records that the log is storing at a point in time five minutes from the time the consumer client was invoked

The client is implemented as a *push consumer*, able to passively accept events pushed to it by the log's event channel (see *The Client as an Event Supplier* on page 40). In many ways, the consumer client is similar to the supplier:

- they both implement `org.omg.CosEventComm`'s callback operations. Notwithstanding that the consumer client is a *push consumer* and not a *push supplier*, it implements the `PushConsumerOperations` instead of the

2.3 The NotifyLog and Event-style Events

PushSupplierOperations - and with one additional, important difference: there is an additional callback method, *push()*, which must be implemented and which receives the events transmitted by the channel.

- they both use a proxy to communicate with the log; the proxy, a *ProxyPushSupplier* in the consumer's case, is obtained using methods similar to those that were used by the supplier client
- they both use the Naming Service to locate the log object

Import Statements

The program uses the services and associated APIs listed below. Applications or classes that use these services or APIs must include the import statements shown below.

ORB

```
com.prismt.orb.ObjectAdapter
```

NotifyLog and for Time values

```
org.omg.DsLogAdmin.*
org.omg.DsNotifyLogAdmin.*
org.omg.CosNotification.EventType
org.omg.CosNotifyComm.*
org.omg.CosNotifyChannelAdmin.*
java.util.GregorianCalendar
java.util.Date
```

Naming Service

```
org.omg.CosNaming.*
org.omg.CosNaming.NamingContextPackage.*
org.omg.CosNaming.NamingContextExt.*
org.omg.CosNaming.NamingContextExtPackage.*
```

Consumer Client Initialisation

This client declares approximately the same type of objects and variables as declared for the supplier client, with the exception that *Timer* is not used - since the *PushConsumerOperations.push()* method (implemented by the client) receives events whenever they arrive. (Compare this with the supplier client initialisation code shown under *Supplier Client Initialisation* on page 38).

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

NotifyLog log = null;
```

```

org.omg.CORBA.Any event;
ProxyPushSupplier supplierProxy;

NamingContextExt rootContext = null;
NameComponent notifyLog[];

private final long duration = 3 * 60 * 1000; // 3 minutes in millisecs

```

The following code example shows, as in the supplier client example, that the ORB is initialised, the client is declared as a transient CORBA object, a reference to the NotifyLog object is obtained using the Naming Service, and the client makes itself available to receive requests.

```

orb = ObjectAdapter.init(args);

// instantiate this client as a transient CORBA object
ObjectAdapter.createTransient(this);

// obtain reference to log using Naming Service
obj = orb.resolve_initial_references("NameService");
rootContext = NamingContextExtHelper.narrow(obj);
notifyLog = new NameComponent[1];
notifyLog[0] = new NameComponent("NotifyLogA", "log");
log = NotifyLogHelper.narrow(rootContext.resolve(notifyLog));

// establish this object as a push consumer and
// connect to the log's proxy supplier
PushConsumer consumer =
    PushConsumerHelper.narrow(ObjectAdapter.getObject(this));

ConsumerAdmin consumerAdmin = log.default_consumer_admin();
org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder();

ProxySupplier proxy =
    consumerAdmin.obtain_notification_push_supplier (
        ClientType.ANY_EVENT, id);

supplierProxy = ProxyPushSupplierHelper.narrow(proxy);
supplierProxy.connect_any_push_consumer(consumer);

// enable incoming requests, without blocking, for log callbacks
ObjectAdapter.ready(false);

```

After the client is established as a *PushConsumer*, it is connected to the log's supplier proxy using:

```

PushConsumer consumer =
    PushConsumerHelper.narrow(ObjectAdapter.getObject(this));

ConsumerAdmin consumerAdmin = log.default_consumer_admin();
org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder();

ProxySupplier proxy =
    consumerAdmin.obtain_notification_push_supplier (
        ClientType.ANY_EVENT, id);

supplierProxy = ProxyPushSupplierHelper.narrow(proxy);
supplierProxy.connect_any_push_consumer(consumer);

```

The Client as an Event Consumer

Similarly, as for the clients which are supplying the events, the consumer clients connect to an event channel through a proxy using the *push* model¹. The steps required to establish an event consumer are the same as those for establishing an event supplier (see *The Client as an Event Supplier* on page 40), except that:

- a *ConsumerAdmin* object is used to obtain a *supplier* proxy
- the consumer connects to the *supplier* proxy using `obtain_push_supplier()` for the push model and `obtain_pull_supplier()` for the pull model
- the *push()* callback method is the method that will retrieve events when using the push model - the developer must implement this method to suit their particular requirements

Receiving Events

The consumer client's *push()* method implementation, shown below, receives the events which have been forwarded by the NotifyLog.

```
// PushConsumerOperations callback methods
public void disconnect_push_consumer()
{
    System.out.println ("Disconnected by proxy.");
} // end disconnect_push_consumer

public void push(org.omg.CORBA.Any event)
{
    System.out.println("Event received contains: " + event.extract_long());
} // end push

public void offer_change(EventType[] added, EventType[] removed)
{
    System.out.println ("Consumer offer changed.");
} // end offer_change
```

This particular implementation extracts a *long* from the event using the event's *extract_long()* method, then displays the value.

Note that the log's forwarding state must be set to *on* (which is the default value) in order for the log to forward events to the consumer. If this state is *off*, it can be set to *on* using:

```
log.set_forwarding_state(ForwardingState.on)
```

1. Remembering that the PrismTech implementation only supports the *push* model.

Retrieving Records

The `Log.retrieve()` method (inherited by all log types) retrieves a sequential list of records, based on the time that the first or last record in the list was placed in the record store. Note that `retrieve()` uses Gregorian time.

`retrieve()` takes three parameters (listed in order):

- `from_time` - the Gregorian time, in 100 nanosecond units, from when the first record in the required list was stored
- `how_many` - the number of records to retrieve: if this value is positive, then the records will be retrieve in a positive direction from the oldest record to the newest; if the value is negative, then the newest record will be retrieved first followed by the next newest and so forth in a negative direction
- `i` - an iterator holder which is returned by `retrieve()` reporting the number of records which have been returned; the iterator can be declared as null if this value is not needed

The code example below retrieves the latest ten records which have been stored in the log: the `from_time` parameter is set to the current time and the `how_many` parameter is set to `-10`. This will retrieve records starting with the record which was stored at the current time and then work backwards, retrieving successively older records. The retrieved records will be held in a `LogRecord` array.

```
// array for retrieved records
LogRecord[] records;

IteratorHolder iter = new IteratorHolder();

// establish current time and convert ms to 100 ns
GregorianCalendar calendar = new GregorianCalendar();
long basetime = - calendar.getGregorianChange().getTime();
long currentTime = (System.currentTimeMillis () + basetime) * 10000;

System.out.println(
    "\nRetrieving the latest 10 records backwards from the current time.");
records = log.retrieve(currentTime, -10, iter);
if (records.length < 1 )
{
    System.out.println("No records retrieved.\n");
}
else
{
    // display information for each retrieved record
    System.out.println ("\n-----\n" +
        "ID\tTime\t\t\t\tInfo");
    for(int i = 0; i < records.length; i++)
    {
        System.out.println(records[i].id +
            "\t" + new Date(records[i].time/10000 - basetime) +
```

2.4 NotifyLog, Structured Events and More

```
        "\t" + records[i].info);  
    }  
    System.out.println ("-----");  
}
```

The client then displays the time the retrieved records were stored, along with their event data (held in the record's *info* field).

Cleanup

Before the client terminates it must destroy the proxy it is connected to and stop receiving requests from the ORB: left-over proxies which are not destroyed contribute to resource leakage.

```
supplierProxy.disconnect_push_supplier();  
ObjectAdapter.shutdown();
```

Exceptions

The example code shown above, which creates a NotifyLog consumer client and uses a Naming Service name binding, must either throw or catch the exceptions listed below.

ORB

```
org.omg.CORBA.ORBPackage.InvalidName
```

NotifyLog

```
org.omg.CosNotifyChannelAdmin.AdminLimitExceeded  
org.omg.CosEventChannelAdmin.TypeError  
org.omg.CosEventChannelAdmin.AlreadyConnected
```

Naming Service

```
org.omg.CosNaming.NamingContextPackage.InvalidName  
org.omg.CosNaming.NamingContextPackage.CannotProceed  
org.omg.CosNaming.NamingContextPackage.NotFound
```

2.4 NotifyLog, Structured Events and More

This topic introduces some of the NotifyLog's more advanced features and explains how to:

1. Set Quality of Service (QoS) and Administration properties not covered in Section 2.3, *The NotifyLog and Event-style Events*.
2. Use a log filter for storing only selected records.
3. Use a filter on the log's notification channel to filter events sent to consumer clients.

4. Implement a supplier client which
 - a) creates and sends structured events to the log
 - b) sets a QoS property for its proxy.
5. Implement a consumer client which can
 - a) receive structured events forwarded by the log
 - b) retrieve information held in the NotifyLog's record store.
6. Monitor events generated by the log and log factory.

Creating a NotifyLog

This example shows how to create a NotifyLog which will use structured events. This example extends information provided in the previous BasicLog and NotifyLog topics, especially regarding items which are specific to using structured events.

Import Statements

The example NotifyLog creation program uses the services and associated APIs listed below. Applications or classes that use these services or APIs must include the import statements shown below.

ORB

```
com.prismt.orb.ORBAdapter
```

NotifyLog

```
org.omg.DsLogAdmin.*
org.omg.DsNotifyLogAdmin.*
org.omg.CosNotification.*
org.omg.CosNotifyFilter.Filter;
import org.omg.CosNotifyFilter.FilterFactory;
import org.omg.CosNotifyFilter.ConstraintExp;
import org.omg.CosNotifyFilter.InvalidConstraint;
import org.omg.CosNotifyFilter.InvalidGrammar
```

Naming Service

```
org.omg.CosNaming.*
org.omg.CosNaming.NamingContextPackage.*
org.omg.CosNaming.NamingContextExt.*
org.omg.CosNaming.NamingContextExtPackage.*
```

NotifyLog Initialisation

The ORB object for the program is declared and initialised in the same manner as described in Section 2.2, *The Basics and the BasicLog*, on page 21. Like the previous examples, the Naming Service is used to facilitate location of the log object and therefore Naming Service objects are also declared.

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

org.omg.CORBA.Object namingService = null;
NamingContextExt rootContext = null;
NameComponent notifyLog[];
```

NotifyLog Creation and Configuration

A NotifyLog object, called *log*, is created using the `create_with_id()` method, as similarly shown under *The NotifyLog and Event-style Events* on page 32. Additional QoS and Admin properties to those used in the *The NotifyLog and Event-style Events* are configured and set.

As shown in the following code extract, the log object in this example is assigned an id of `10`. A test is made to determine if a log with an id of `10` already exists: this examples replaces the existing log: a working application may decide to simply change the value of `logId` and create a new, different log.

```
orb = ORBAdapter.init(args);

org.omg.CORBA.Object object =
    orb.resolve_initial_references("NotifyLogFactory");
NotifyLogFactory factory = NotifyLogFactoryHelper.narrow(object);

if(factory == null)
{
    System.out.println("Can not resolve initial reference " +
        "of NotifyLogFactory.");
    System.exit(1);
}

NotifyLog log = null;
int logId = 10;

// If Notify log with specified id exists, remove it, then
// create a new one with an id of 0, which discards oldest events
// when full, has a size limit of 50,000 bytes, and generates
// threshold alarms when 25%, 50%, and 75% capacity reached.
org.omg.CORBA.Object corbaObj = null;
corbaObj = factory.find_log(logId);
if ( corbaObj != null )
{
    System.out.println("Existing log found and being replaced.");
    log = NotifyLogHelper.narrow(corbaObj);
    log.destroy();
}
```

The QoS and Admin properties are now defined using two `Property` arrays, `qos` and `admin`:

- the `ConnectionReliability` and `EventReliability` properties are both set to `Persistent` to ensure the reliability of the notify channel connection and that events can be recovered in case of a system failure
- `MaxQueueLength` is set to limit the channel's event queue buffer to hold a maximum of 100 events
- `MaxConsumers` is set to limit (to 10) the maximum number of consumers that can be connected at one time to the log

```
// Log QoS set automatically to guarantee log availability
// Notification QoS set to guarantee connection and
// event reliability
Property[] qos = new Property[2];
qos[0] = new Property();
qos[0].name = ConnectionReliability.value;
qos[0].value = orb.create_any();
qos[0].value.insert_short(Persistent.value);

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

// Notification Admin set to have a maximum queue length of
// 100 events and a maximum of 10 consumers
Property[] admin = new Property[2];
admin[0] = new Property();
admin[0].name = MaxQueueLength.value;
admin[0].value = orb.create_any();
admin[0].value.insert_long(100);

admin[1] = new Property();
admin[1].name = MaxConsumers.value;
admin[1].value = orb.create_any();
admin[1].value.insert_long(10);

// set values for threshold alarms
short[] thresholds = {25, 50, 75};

// create the log
log = factory.create_with_id(logId, halt.value, 50000,
                             thresholds, qos, admin);
```

The log record size, threshold levels, and log full behaviour are set.

Note that the `halt` behaviour specified when the record store is full requires clients, which supply events to the log, to test for a log full condition (using `Log.get_availability_status().log_full`): an exception is thrown if a supplier attempts to send an event to a log when the record store is full and `halt` has been specified.

The remaining log properties can now be set. The following code sets the maximum record life to five hours (in units of one second) and ensures that the log will forward events to consumers.

```
// enable log forwarding and set the maximum log record lifetime
// to 5 hours in seconds - prevent any logging while doing so
log.set_administrative_state(AdministrativeState.locked);
```

```
log.set_forwarding_state(ForwardingState.on);  
log.set_max_record_life (60 * 60 * 5);
```

Note that the log's administrative state has not yet been unlocked, since a filter will be added to the log: the administrative state will be unlocked after the filter has been added.

Filtering

Structured events which are sent to NotifyLogs can be filtered in order to remove unwanted events. Filtering can be used to control event transmission and to improve overall performance. For example, unwanted events

- can be stopped from entering the event channel, thereby improving the log channel's effective bandwidth and performance
- can be prevented from being placed in the record store, thereby conserving resources
- can be filtered out on a consumer-by-consumer basis, providing fine-grained control and performance tuning on the client-side of the log's event channel

Logs are able to filter events in one of two ways: on the event channel or immediately before events are received by the record store, which are referred to here, respectively, as *channel-based* filtering or *log-based* filtering. Both of these methods are briefly described below.



Example code for performing log-based filtering is provided as a part of the example NotifyLog creation program. Example code for performing channel-based filtering is provided under *Creating a Consumer Client for the NotifyLog* on page 64, as a part of the example consumer program, and demonstrates how to create a filter on a consumer-by-consumer basis.

Filtering support is provided by the *CosNotifyFilter* module and related interfaces, inherited from the Notification Service. Developers who use filters should have a good understanding of *CosNotifyFilter* and should refer to the sections in the OMG's *Notification Service Specification* and the OpenFusion *Notification Service Guide* which describe the *Filter*, *FilterFactory*, and *FilterAdmin* interfaces.¹

1. In addition to the standard Filter interface described here, there is also a *MappingFilter* interface: information on Mapping Filters is provided in the referenced texts.

Although a detailed discussion of filtering is beyond the scope of this manual, basic information and guidance on creating and managing filters are provided on this topic under *Creating a Consumer Client for the NotifyLog* on page 64.

The Filter Object

Filtering is performed with a filter object. A filter can be added to a proxy object, channel admin object, or to a log itself. 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 memory leakage. However, care must be taken when destroying filter objects that are used by multiple admin/proxy/log objects in order to avoid inadvertently destroying a filter which is still in use.

Filter Location

The events which are filtered depend on where the filter is added, as follows:

- *consumer proxy* (connects to a supplier) - filters the events received directly from the supplier and therefore events sent to the channel, log, and all supplier proxies (and their associated consumers) are filtered
- *supplier proxy* (connects to a consumer) - filters the events received from the channel before being sent to the proxy's consumer
- *supplier admin* - filters all events received from all consumer proxies connected to the supplier admin and therefore filters events sent to the log and all supplier proxies
- *consumer admin* - filters all events received from the channel and therefore filters events sent to any supplier proxies connected to the consumer admin
- *log* - only filters the events received from the channel which are sent to the log's record store

Main Filter Components

The main components of a filter are an array of constraint expressions and a set of methods which are used to manage the filter.

A constraint expression defines the criteria by which events are filtered. The expression is written using the Notification Service's constraint language: the service's default grammar is the *Extended Trader Constraint Language*, ETCL. Filtering is performed on the *event header* and *filterable body* components of structured events.

The essential filter methods to be familiar with include:

- those which add, modify, and remove constraints, i.e.

```
- add_constraints()  
- modify_constraints()  
- get_constraints()  
- get_all_constraints()  
- remove_all_constraints()
```

- the *destroy()* method which destroys the filter object

Filter Creation

Filters are created as follows:

- Step 1:** Obtain a reference to a `FilterFactory` from an event channel or the log using its `default_filter_factory()` method:

```
FilterFactory factory = Admin.default_filter_factory();
```

or

```
FilterFactory factory = log.default_filter_factory();
```

- Step 2:** Create the filter using the factory's `create_filter()` method, passing it a string defining the grammar to be used - this is normally `default_grammar.value`, which is the Notification Service's default grammar, "EXTENDED_TCL":

```
Filter filter =  
    factory.create_filter(default_grammar.value).
```

- Step 3:** Create one or more constraint expressions, of type `ConstraintExp`, then create a `ConstraintExp` array containing the constraints. A constraint expression consists of two elements:

- a two element array, of type `EventType`, containing the `domain_name` and `type_name` components of the structured event's fixed header - this identifies the type of event to be filtered

- an expression, using the same grammar that was passed to `create_filter()`, which defines the filtering criteria, e.g. `"$.switch == 'open'"`, which means that a property called `switch` must contain a value of `'open'` in order to be passed through the filter

For example, a simple constraint expression could be constructed as follows:

```
EventType[] type = {"teleco", "mobile"};
ConstraintExp exp =
    new ConstraintExp(type,
        "$.filterable_data(switch) == 'open'");

ConstraintExp[] expressions = {exp};
```

This would allow structured events to pass through the filter which have a fixed header with a `domain_name` of `'telco'`, `type_name` of `'mobile'`, and a filterable body property called `switch` which contains a value of `'open'`.

Step 4: Add the constraints array to the filter using its `add_constraints()` method:

```
filter.add_constraints(expressions);
```

Step 5: Associate the filter to the required object(s), noting that:

- if the filter is to be used as a channel-based filter, i.e. associated with an `admin` or `proxy` object, then the `add_filter()` method should be used:

```
FilterID id = Admin.add_filter(filter);
```

and where `add_filter()` returns a unique id (a long); this id is needed by the `remove_filter()` and `get_filter()` methods.

- if the filter is to be used as a log-based filter, i.e. filter the events sent *only* to the record store, then the `set_filter()` method should be used:

```
log.set_filter(filter);
```

The following code example demonstrates how to create a log-based filter.

```
// filter out all events sent to the record store except where
// the "number" property is an even number (value of 0)
FilterFactory filterFactory = log.default_filter_factory();
Filter filter = filterFactory.create_filter(default_grammar.value);

EventType type = new EventType("software", "example");
EventType[] types = {type};
ConstraintExp exp =
    new ConstraintExp(types, "$.filterable_data(number) == 0");

ConstraintExp[] expressions = {exp};
filter.add_constraints(expressions);
log.set_filter(filter);
```

```
log.set_administrative_state(AdministrativeState.unlocked);
```

(Note that the code shows the log's administrative state being unlocked after setting the filter since configuration of the log is complete at this stage.)

Filter Removal and Destruction

Removing a filter disassociates the filter from an object. Removing a filter does not destroy it - the filter can still be referenced by other objects which the filter has been added to.

Destroying a filter actually destroys it and releases any resources used by the filter. No object will be able to use the filter after the filter has been destroyed.

- A filter is removed by calling the `remove_filter()` method from the object it is to be removed from, e.g. `Admin.remove_filter(id)`, where `id` is the `FilterID` of the filter to be removed.
- A filter is destroyed by calling its `destroy()` method, e.g. `filter.destroy()`.

A filter does not need to be removed from objects before it is destroyed, i.e. it is not a prerequisite to call `remove_filter()` before calling `destroy()`.

Registering with the Naming Service

Continuing with the log creation example, the final operation is to bind the log instance to a name using the Naming Service. The `id` component of the name is defined as "NotifyLog" in order to distinguish it from the other logs used in the examples.

```
// register the log object with the Naming Service
namingService = orb.resolve_initial_references ("NameService");
rootContext = NamingContextExtHelper.narrow(namingService);
notifyLog = new NameComponent[1];
notifyLog[0] = new NameComponent("NotifyLogB", "log");
rootContext.rebind(notifyLog, log);
System.out.println("NotifyLogB created");
```

NotifyLog Exceptions

The example code shown above must either throw or catch the exceptions listed below.

ORB (and connecting to services)

```
org.omg.CORBA.ORBPackage.InvalidName
```

NotifyLog

```
org.omg.DsLogAdmin.LogIdAlreadyExists
org.omg.DsLogAdmin.InvalidLogFullAction
org.omg.DsLogAdmin.InvalidThreshold
org.omg.CosNotification.UnsupportedAdmin
org.omg.CosNotification.UnsupportedQoS
org.omg.CosNotifyFilter.InvalidGrammar
org.omg.CosNotifyFilter.InvalidConstraint
```

Naming Service

```
org.omg.CosNaming.NamingContextPackage.InvalidName
org.omg.CosNaming.NamingContextPackage.NotFound
org.omg.CosNaming.NamingContextPackage.CannotProceed
```

Creating a Supplier Client for the NotifyLog

The supplier client example for the NotifyLog shows how to:

- establish a *structured push supplier* client which can send structured events to a NotifyLog
- set a Quality of Service (QoS) property for the client's proxy
- create a structured event which contains header information which can be filtered by the log and consumer clients
- handle the situation where the log's record store is full and refuses to store events

The structure of this supplier client example is similar to the supplier client for the example that uses event-style events: events are created and sent at regular intervals for a fixed duration (using the `Timer` and `TimerTask` thread mechanism). Note that the creation of the structured event varies in many ways from the creation of the simple event-style event.

Import Statements

The program uses the services and associated APIs listed below. Applications or classes that use these services or APIs must include the import statements shown below.

ORB

```
com.prismt.orb.ObjectAdapter
```

NotifyLog

```
org.omg.DsLogAdmin.*
org.omg.DsNotifyLogAdmin.*
org.omg.CosNotifyComm.*
org.omg.CosNotifyChannelAdmin.*
org.omg.CosNotification.*
com.prismt.cos.CosNotification.NotificationExtensions.*
```

Naming Service

```
org.omg.CosNaming.*
org.omg.CosNaming.NamingContextPackage.*
org.omg.CosNaming.NamingContextExt.*
org.omg.CosNaming.NamingContextExtPackage.*
```

J2SE'S Timer and TimerTask

```
java.util.Timer.*
java.util.TimerTask
```

Supplier Client Initialisation

As with the previous examples, objects and variables are declared which are necessary to connect to the ORB, log, proxy, Naming Service, etc. Note that since this client will be sending structured events using the push model, a reference to a structured push consumer proxy, a *StructuredProxyPushConsumer*, will therefore need to be obtained.

```
// structured push suppliers must implement StructuredPushSupplierOperations
public class NotifySupplierB implements StructuredPushSupplierOperations
{
    org.omg.CORBA.ORB orb = null;
    org.omg.CORBA.Object obj = null;

    NotifyLog log = null;
    StructuredProxyPushConsumer pushConsumerProxy = null;

    NamingContextExt rootContext = null;
    NameComponent notifyLog[];

    private Timer sendTimer;

    private final long duration = 3 * 60 * 1000; // activity time in millisecs
    private final long interval = 1000; // 1 second
```

Also, as in the previous examples, a reference to the ORB is obtained and the client is instantiated as a transient CORBA object:

```
orb = ObjectAdapter.init(args);

// instantiate this client as a transient CORBA object
ObjectAdapter.createTransient(this);
```

A reference to the desired NotifyLog instance is obtained using the Naming Service, as before:

```
// obtain reference to log using Naming Service
obj = orb.resolve_initial_references ("NameService");
rootContext = NamingContextExtHelper.narrow(obj);
notifyLog = new NameComponent[1];
notifyLog[0] = new NameComponent("NotifyLogB", "log");
log = NotifyLogHelper.narrow(rootContext.resolve(notifyLog));
```

The client will now establish itself as a structured push supplier and connect to the consumer proxy, as shown in the code given below. Note that Notification Service's API is used to do this. Briefly, the procedure accommodates the features of the Notification Service, such as the provision of multiple channels, ability to handle different event types and transmission models (resulting in numerous proxy types), plus provision of Quality of Service. In particular, note that

- the NotifyLog's *default* supplier admin object is obtained (using `default_supplier_admin()`), since NotifyLog can have multiple channels and admin objects, and
- the StructuredProxyPushConsumer which will be used is obtained by narrowing the more general ProxyConsumer proxy and using `ClientType.STRUCTURED_EVENT` to identify which type of proxy will be created.

The example code also sets the `MaxInactivityInterval` for the proxy in order to demonstrate how to set a QoS property for a proxy. Although it is not necessary to set QoS properties when connecting to a proxy, setting this particular QoS property is useful as a mechanism for destroying a proxy when its associated client does not or can not explicitly destroy the proxy itself, such as when the client is terminated by a **Ctrl-C** executed on the command line, i.e. this mechanism can prevent the service filling up with 'dead' proxies.

`MaxInactivityInterval` destroys the proxy when there has been no activity detected by the client after a fixed time interval. The time-out for the interval is reset each time client activity is detected.

```
// establish this object as a structured push supplier
StructuredPushSupplier supplier =
    StructuredPushSupplierHelper.narrow(ObjectAdapter.getObject(this));

// obtain the log's structured push consumer proxy and connect to it
SupplierAdmin supplierAdmin = log.default_supplier_admin();
org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder();

ProxyConsumer proxyConsumer =
    supplierAdmin.obtain_notification_push_consumer(
        ClientType.STRUCTURED_EVENT, id);
```

2.4 NotifyLog, Structured Events and More

```
pushConsumerProxy =
    StructuredProxyPushConsumerHelper.narrow(proxyConsumer);

// set proxy's MaxInactivityInterval QoS to timeout after
// 20 seconds if this supplier is prematurely terminated
Property[] qos = new Property[1];
qos[0] = new Property();
qos[0].name = MaxInactivityInterval.value;
qos[0].value = orb.create_any();
qos[0].value.insert_ulongLong(10000 * 1000 * 5);

pushConsumerProxy.set_qos(qos);

// connect to the proxy
pushConsumerProxy.connect_structured_push_supplier(supplier);

// enable incoming requests, without blocking, for log callbacks
ObjectAdapter.ready(false);
```

After the client is connected to the proxy and the QoS properties set, the ORB is notified that it is able to accept callbacks using `ObjectAdapter.ready(false)`. The supplier is now ready to create and send structured events.

Structured Events in Brief

A structured event consists of two main parts:

- an *event header* which contains a *fixed header* and *variable header*, these contain:
 - *event domain* (*domain_name*) - the domain of a particular vertical industry where the event type is defined, such as *telecommunications*, *finance*, *transportation*, etc.
 - *event type* (*type_name*) - the type of particular event within the domain, for example *StockQuote* within the *finance* domain
 - *event name* (*event_name*) - a unique name for the particular event instance being transmitted
 - QoS property settings as a list of *name-value* pairs
- an *event body* containing
 - a *filterable_body* which is a list of *name-value* pairs which are used to filter the event
 - a *remainder_of_body* which is the event itself, an *Any*

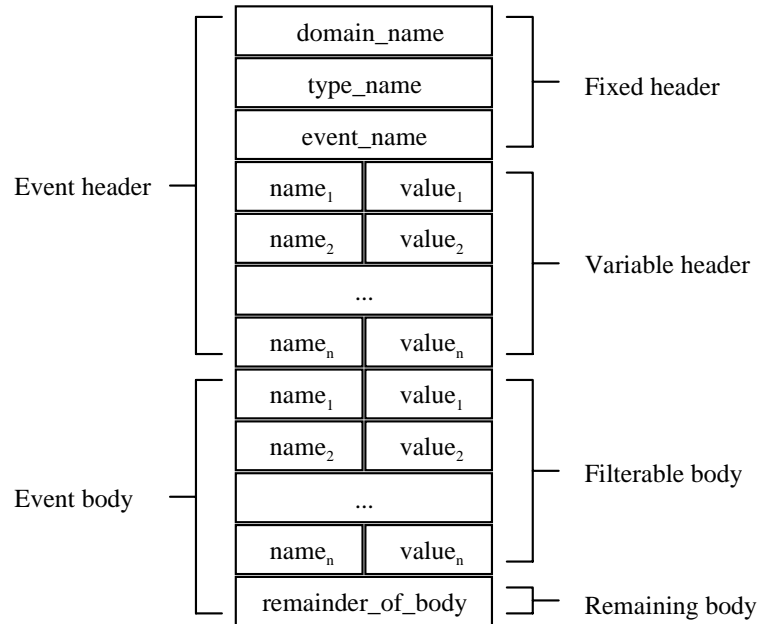


Figure 6 Structured Event

Creating and Sending the Events

The following code example is the client's implementation of the `TimerTask`'s `run()` method. This method creates and sends a series of structured events to the `NotifyLog` instance for a fixed duration (determined by comparing the value returned by `System.currentTimeMillis()` with the value held in `stopSendingTime`).

The code example shows how to:

- define the domain name ("software"), event type ("example") and event name ("Notify Log") fields of the structured event's fixed header
- create and define two filterable properties ("number" and "time") in the filterable body header
- define and add an event to the event's `remainder_of_body`
- send the event to the proxy

The fixed header will be used by the log filter and consumer client's proxy to identify the event; the properties in the filterable body will be used to filter the event.

Points to Note

- A check is made, before an event is created and sent, to ensure that the log's record store is *not full* - this is in addition to checking that the log is not *off duty*. This additional check, that the log is not full, is necessary since the particular log instance which will receive the events is configured to reject events when the record store is full: this configuration will cause a *LogFull* exception to be thrown when trying to store an event when the store is full: using `log.get_availability_status().log_full` in the `if()` expression avoids this exception being thrown.
- In order to provide an example mechanism for filtering, a counter is incremented each time an event is sent: if the value of the counter is even, then a *0* (zero) is assigned to the event's *number* property (held in the event's filterable body); if the counter is odd, a *1* is assigned. The log and consumer will use these values to filter the events:

```
counter++;
num = counter % 2;
...
event.filterable_data[0] = new Property();
event.filterable_data[0].name = "number";
event.filterable_data[0].value = orb.create_any();
event.filterable_data[0].value.insert_long(num);
```

- `pushConsumerProxy.disconnect_structured_push_consumer()` is called when `run()` has stopped sending events; this is used to destroy the proxy (since it is no longer needed).

```
public void run()
{
    // check timeout and record store availability
    if((System.currentTimeMillis() < stopSendingTime) &&
        (log.get_availability_status().off_duty == false) &&
        (log.get_availability_status().log_full == false))
    {
        try
        {
            counter++;
            num = counter % 2; // determine if counter is odd or event
            // create the structured event to be sent to the log
            StructuredEvent event = new StructuredEvent();

            // create header
            EventType type = new EventType("software", "example");
            FixedEventHeader fixedHeader =
                new FixedEventHeader(type, "Notify Log");
            Property variableHeader[] = new Property[0];
            event.header = new EventHeader(fixedHeader, variableHeader);
```



```

// create a filterable body containing properties to hold
// the event's number and the time it was sent
event.filterable_data = new Property[2];

event.filterable_data[0] = new Property();
event.filterable_data[0].name = "number";
event.filterable_data[0].value = orb.create_any();
event.filterable_data[0].value.insert_long(num);

event.filterable_data[1] = new Property();
event.filterable_data[1].name = "time";
event.filterable_data[1].value = orb.create_any();

// add a message to the body
event.remainder_of_body = orb.create_any();
event.remainder_of_body.insert_string("Your Message Here.");

// add the time that event is sent
event.filterable_data[1].value.insert_ulonglong(
    System.currentTimeMillis());

// send the event
pushConsumerProxy.push_structured_event(event);

System.out.println ("The number value of the event sent is " +
    event.filterable_data[0].value.extract_long() + ".");
}
catch (org.omg.CosEventComm.Disconnected ex)
{
    System.out.println("Disconnected");
    System.exit(1);
}
}
else
{
    if(log.get_availability_status().off_duty)
    {
        System.out.println("The log is off duty " +
            "- cannot log event.");
    }
    if(log.get_availability_status().log_full)
    {
        System.out.println("The log is full " +
            "- cannot log event.");
    }
    if(System.currentTimeMillis() >= stopSendingTime)
    {
        System.out.println("Supplier finished sending events.\n");
    }

    // cleanup - destroy proxy, stop incoming requests
    pushConsumerProxy.disconnect_structured_push_consumer();
    ObjectAdapter.shutdown();

    // stop thread
    sendTimer.cancel();
}
} // end run

```

Supplier Client Exceptions

The example code shown above must either throw or catch the exceptions listed below.

ORB

org.omg.CORBA.ORBPackage.InvalidName

NotifyLog

org.omg.CosEventChannelAdmin.AlreadyConnected
org.omg.CosEventComm.Disconnected
org.omg.CosNotifyChannelAdmin.AdminLimitExceeded
org.omg.CosNotification.UnsupportedQoS

Naming Service

org.omg.CosNaming.NamingContextPackage.InvalidName
org.omg.CosNaming.NamingContextPackage.NotFound
org.omg.CosNaming.NamingContextPackage.CannotProceed

Creating a Consumer Client for the NotifyLog

The consumer client sets a filter on the proxy which connects it to the log's notify channel, receives events which are allowed to pass through the filter, and retrieves the last ten records held in the log's record store.

This example consumer is similar to the supplier in terms of obtaining a connection to the log, and similar to the log creation example in terms of creating a filter. Since this example builds on previous ones, only the relevant differences will be shown or highlighted.

Import Statements

The program uses the services and associated APIs listed below. Applications or classes that use these services or APIs must include the import statements shown below.

OpenFusion Orb Adaptor

com.prismt.orb.ObjectAdapter

NotifyLog and for Time values

*org.omg.DsLogAdmin.**
*org.omg.DsNotifyLogAdmin.**
*org.omg.CosNotifyComm.**
*org.omg.CosNotifyChannelAdmin.**
*org.omg.CosNotification.**

org.omg.CosNotifyFilter.Filter
org.omg.CosNotifyFilter.FilterFactory

```
org.omg.CosNotifyFilter.ConstraintExp
org.omg.CosNotifyFilter.InvalidConstraint
org.omg.CosNotifyFilter.InvalidGrammar
```

```
java.util.GregorianCalendar
java.util.Date
```

Naming Service

```
org.omg.CosNaming.*
org.omg.CosNaming.NamingContextPackage.*
org.omg.CosNaming.NamingContextExt.*
org.omg.CosNaming.NamingContextExtPackage.*
```

Consumer Client Initialisation

The consumer performs similar initialisation steps to the previous examples, and as shown in the code examples below. Note that the consumer uses a proxy of type `StructuredProxyPushSupplier` (which is obtained by narrowing a `ProxySupplier` with a client type of `STRUCTURED_EVENT`).

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

NotifyLog log = null;

StructuredProxyPushSupplier pushSupplierProxy = null;
Filter filter = null;

org.omg.CORBA.Any event;

NamingContextExt rootContext = null;
NameComponent notifyLog[];

private final long duration = 3 * 60 * 1000; // activity time in millisecs

orb = ObjectAdapter.init(args);

// instantiate this client as a transient CORBA object
ObjectAdapter.createTransient(this);

// obtain reference to log using Naming Service
obj = orb.resolve_initial_references("NameService");
rootContext = NamingContextExtHelper.narrow(obj);
notifyLog = new NameComponent[1];
notifyLog[0] = new NameComponent("NotifyLogB", "log");
log = NotifyLogHelper.narrow(rootContext.resolve(notifyLog));

// establish this object as a structured push consumer
StructuredPushConsumer consumer =
    StructuredPushConsumerHelper.narrow(ObjectAdapter.getObject(this));

// obtain the log's structured push supplier proxy and connect to it
ConsumerAdmin consumerAdmin = log.default_consumer_admin();
org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder();

ProxySupplier proxySupplier =
```

```
consumerAdmin.obtain_notification_push_supplier(  
    ClientType.STRUCTURED_EVENT, id);  
  
pushSupplierProxy =  
    StructuredProxyPushSupplierHelper.narrow(proxySupplier);
```

Setting a Proxy Filter

After obtaining the proxy, the client creates and adds a filter to it. This filter is configured to only accept events which have a property called *number* (in the filterable body) which has a value of *1* (i.e. odd numbered events - see *Points to Note* on page 62).

Points to Note

- A filter factory is used to create the filter; this factory is obtained from the log.
- the *domain name* and *type name* of the events to be filtered are used to create an `EventType` object, called *type* in the example. The `EventType` is added to an `EventType` array, which is used by a `ConstraintExp` object to identify which events to filter. A constraint expression, also used by the `ConstraintExp` object, defines the type of *header*, *property name*, and *condition* used to test the value of the property.
- The `add_filter()` method is used to add the filter to the proxy, and *not* the `set_filter()` method which is used for an log-based filter

```
try  
{  
    // set filter on proxy to only accept events where  
    // the "number" property is an odd number (value of 1)  
    FilterFactory filterFactory = log.default_filter_factory();  
    filter = filterFactory.create_filter(default_grammar.value);  
  
    EventType type = new EventType("software", "example");  
    EventType[] types = {type};  
    ConstraintExp exp =  
        new ConstraintExp(types, "$.filterable_data(number) == 1");  
  
    ConstraintExp[] expressions = {exp};  
    filter.add_constraints(expressions);  
    pushSupplierProxy.add_filter(filter);  
}  
catch (org.omg.CosNotifyFilter.InvalidGrammar ex)  
{  
    System.out.println("InvalidGrammar");  
    System.exit(1);  
}  
catch (org.omg.CosNotifyFilter.InvalidConstraint ex)  
{  
    System.out.println("InvalidConstraint");  
    System.exit(1);  
}
```

The client is now ready to receive events after having configured and added the filter to the proxy.

Receiving the Structured Events

The NotifyLog consumer receives events similarly to the previous consumer example that used event-style events, noting the differences in the proxy type and version of connection methods.

```
// connect to the proxy
pushSupplierProxy.connect_structured_push_consumer(consumer);
```

The client also ensures that the forwarding state for the log is enabled.

```
// ensure that log is enabled to forward records
ForwardingState state = log.get_forwarding_state();
if (state == ForwardingState.off)
{
    log.set_forwarding_state(ForwardingState.on);
}

// thread for receiving events
// events are received via push_structured_event(),
// the PushConsumerOperations callback method
System.out.println("Will receive structured events for " +
    duration / 60000 + " minutes from now.");
synchronized(this)
{
    try
    {
        this.wait(duration);
    }
    catch (java.lang.InterruptedException ex)
    {
        return;
    }
}

System.out.println("Finished getting structured events.");
```

The `push_structured_event` callback method is responsible for processing any events which are received from the supplier via the log's notify channel. The example implementation extracts the data values held in the filterable data part of received events, as well as extracting the data held in the `remainder_of_body`. The only events which should be received by `push_structured_event` will be those whose `number` property has a value of 1.

```
// StructuredPushConsumerOperations callback methods
public void disconnect_structured_push_consumer()
{
    System.out.println ("Disconnected by proxy");
} // end disconnect_push_consumer

public void push_structured_event(
    org.omg.CosNotification.StructuredEvent event)
{
    Property[] filterable_data;
```

2.4 NotifyLog, Structured Events and More

```
org.omg.CORBA.Any remainder_of_body = orb.create_any();

filterable_data = event.filterable_data;
int len = filterable_data.length;

System.out.println("\nNotifyConsumer received event containing:");
for( int j = 0 ; j < len ; j++ )
{
    if(filterable_data[j].name.equals("time"))
    {
        System.out.println("  filterable_data[" + j + "] " +
            "time: " +
            new Date((long)filterable_data[j].value.extract_ulonglong()).
                toString());
    } else
    {
        System.out.println("  filterable_data[" + j + "] " +
            filterable_data[j].name + ": " +
            filterable_data[j].value);
    }
}

remainder_of_body = event.remainder_of_body ;
System.out.println("  remainder_of_body = " + remainder_of_body);
} // end push

public void offer_change(EventType[] added, EventType[] removed)
{
    System.out.println ("Consumer offer changed");
} // end offer_change
```

Retrieving Log Records

The client *retrieves* the log records as structured events, in addition to *receiving* the events sent through the log's channel. The following example shows how the structured event records are retrieved and information extracted from them. The value of the filterable body's (*filterable_data*) *number* property should be 0, since the filter for the log's record store was set to only accept events containing this value in the *number* property.

```
// array for retrieved records
LogRecord[] records;
IteratorHolder iter = new IteratorHolder();

// establish current time and convert ms to 100 ns
GregorianCalendar calendar = new GregorianCalendar();
long basetime = - calendar.getGregorianCalendar().getTime();
long currentTime = (System.currentTimeMillis () + basetime) * 10000;

System.out.println(
    "\nRetrieving the latest 10 records backwards from the current time.");
records = log.retrieve(currentTime, -10, iter);
if (records.length < 1)
{
    System.out.println ("No records retrieved.\n");
}
else
{
    // display information for each retrieved record
    System.out.println ("\n-----" +
        "Retrieved records listed below.");
    for(int i = 0; i < records.length; i++)
    {
```

```

org.omg.CORBA.Any info = orb.create_any();
StructuredEvent se;
Property[] filterable_data;
org.omg.CORBA.Any remainder_of_body = orb.create_any();

se = StructuredEventHelper.extract(records[i].info);

filterable_data = se.filterable_data;
int len = filterable_data.length;

System.out.println("\nrecord id: " + records[i].id);

for( int j = 0 ; j < len ; j++ )
{
    if(filterable_data[j].name.equals("time"))
    {
        System.out.println("    filterable_data[" + j + "] " +
            "time: " +
            new Date((long)filterable_data[j].value.extract_ulonglong()).
                toString());
    } else
    {
        System.out.println("    filterable_data[" + j + "] " +
            filterable_data[j].name + ", " +
            filterable_data[j].value);
    }
}

remainder_of_body = se.remainder_of_body;
System.out.println("    remainder_of_body = " +
    remainder_of_body);
}
System.out.println ("-----");
}

```

Cleanup

When the client is finished receiving events, it destroys the proxy's filter, disconnects from (and destroys) the proxy, then notifies the ORB it is unable to accept requests.

```

public void cleanup()
{
    filter.destroy();
    pushSupplierProxy.disconnect_structured_push_supplier();
    ObjectAdapter.shutdown();
}

```

Consumer Client Exceptions

The example code shown above must either throw or catch the exceptions listed below.

ORB

```
org.omg.CORBA.ORBPackage.InvalidName
```

NotifyLog

```
org.omg.CosEventChannelAdmin.AlreadyConnected
org.omg.CosNotifyChannelAdmin.AdminLimitExceeded
```

```
org.omg.CosEventChannelAdmin.TypeError
```

Naming Service

```
org.omg.CosNaming.NamingContextPackage.InvalidName  
org.omg.CosNaming.NamingContextPackage.NotFound  
org.omg.CosNaming.NamingContextPackage.CannotProceed
```

Monitoring Log Generated Events

The following example monitor client shows how events which have been generated by a log and it's log factory can be received and monitored.

The client is similar to a standard NotifyLog consumer client, but with the following notable differences:

- An *event* channel, as opposed to a *notify* channel, is used to transmit and receive the events (note the differences of their relative import statements - see *Import Statements* on page 64 and below): generated events are transmitted as Event-style events (*AnyS*) using an event channel.
- The client connects to the log *factory's* proxy (instead of the log): all generated events, whether from the factory or the log, are transmitted by the factory that created the log.
- The client does not need to obtain a reference to the log object, but to the factory that creates it. Therefore, there is no need to use the Naming Service or other mechanism to locate the log object. In fact, a client may likely be instantiated and active before the log is created: this will occur if the client intends to monitor log creation events generated by the log factory.

Import Statements

The program uses the services and associated APIs listed below. Applications or classes that use these services or APIs must include the import statements shown below.

OpenFusion Orb Adaptor

```
com.prismt.orb.ObjectAdapter
```

NotifyLog and for Time values

```
org.omg.DsLogNotification.*
```

```
org.omg.DsLogAdmin.*
```

```
org.omg.DsNotifyLogAdmin.*
```



```

org.omg.CosNotification.EventType;
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyChannelAdmin.*

java.util.GregorianCalendar
java.util.Date

```

Monitor Client Initialisation

The client implements `PushConsumerOperations` and is initialised similarly to previous NotifyLog client, but note that a reference is obtained to the log factory and not to a log.

```

orb = ObjectAdapter.init(args);

// instantiate this client as a transient CORBA object
ObjectAdapter.createTransient(this);

// obtain reference to log factory
obj = orb.resolve_initial_references("NotifyLogFactory");

NotifyLogFactory factory = NotifyLogFactoryHelper.narrow(obj);
if(factory == null)
{
    System.out.println("\nLog factory not available\n");
    System.exit(1);
}

// establish this object as a push consumer
PushConsumer consumer =
    PushConsumerHelper.narrow(ObjectAdapter.getObject(this));

// obtain the log factory's push supplier proxy and connect to it
org.omg.CORBA.IntHolder id = new org.omg.CORBA.IntHolder();
ProxySupplier proxy =
    factory.obtain_notification_push_supplier(ClientType.ANY_EVENT, id);
pushSupplierProxy = ProxyPushSupplierHelper.narrow(proxy);
pushSupplierProxy.connect_any_push_consumer(consumer);

// set base time for date and time conversion from TimeT
basetime = - new GregorianCalendar().getGregorianChange().getTime();

```

Receiving the Generated Events

The `push()` callback method receives the generated events. The example implementation of this method, shown below, extracts and displays the property values for the various generated events.

```

// StructuredPushConsumerOperations callback methods
public void disconnect_structured_push_consumer()
{
    System.out.println("Disconnected by proxy");
} // end disconnect_push_consumer

```

An important aspect of retrieving generated events is that it may not be possible to determine the property *type* for a particular event in advance, since certain events can contain different types depending on the situation which generated the event. For example, `AttributeValueChange` can contain different types, e.g. a `long` or `omg.org::CORBA::Object`, depending on which attribute has changed, as shown in the output generated by the above code:

```
AttributeValueChange
  id:          10
  time:        Wed Oct 30 13:35:40 GMT 2002
  type:        8
  value type:  long

AttributeValueChange
  id:          10
  time:        Wed Oct 30 13:35:41 GMT 2002
  type:        6
  value type:  omg.org::CORBA::Object
```

Monitor Client Exceptions

The example code shown above must either throw or catch the exceptions listed below.

ORB

org.omg.CORBA.ORBPackage.InvalidName

NotifyLog

org.omg.CosEventChannelAdmin.AlreadyConnected

org.omg.CosNotifyChannelAdmin.AdminLimitExceeded

org.omg.CosEventChannelAdmin.TypeError

3 Supplemental Information

3.1 Exceptions

A brief description of the Log Service exceptions is provided in *Table 3* below. These exceptions are defined in the *DsLogAdmin* module.

i However, there are additional exceptions for those log types which inherit from the Event and Notification Services. These exceptions are described in the *OpenFusion Notification Service Guide*.

Table 3 Log Service Exceptions

Exception	Reason
<i>InvalidParam</i>	Invalid parameter supplied.
<i>InvalidThreshold</i>	Invalid threshold value supplied when setting an alarm threshold. Valid values are expressed as percentages between 0 and 100, non-inclusive.
<i>InvalidTime</i>	The <i>TimeInterval</i> 's or <i>Time24Interval</i> 's <i>stop</i> time is set earlier than the current time, or when a <i>Time24</i> structure contains invalid values.
<i>InvalidTimeInterval</i>	The <i>start</i> time is set later than the <i>stop</i> time.
<i>InvalidMask</i>	The days bit field of a <i>WeekMaskItem</i> contains an illegal value.
<i>LogIdAlreadyExists</i>	Raised when creating a new log and a log with the same id already exists.
<i>InvalidGrammar</i>	The grammar specified is not supported.
<i>InvalidConstraint</i>	The supplied constraint does not conform to the specified grammar and is invalid.
<i>LogFull</i>	An attempt is made to store a record when the record store is full and the log full behaviour is set to <i>halt</i> .
<i>LogOffDuty</i>	An attempt is made to store a record when the log is off duty.
<i>LogLocked</i>	An attempt is made to store a record when the log is locked.

Table 3 Log Service Exceptions (Continued)

Exception	Reason
LogDisabled	An attempt is made to store a record when the log is disabled.
<i>InvalidRecordId</i>	An attempt is made to retrieve or delete a log record that does not exist.
<i>InvalidAttribute</i>	The log attribute is invalid.
<i>InvalidLogFullAction</i>	The specified log full action is not supported.
<i>UnsupportedQoS</i>	The specified QoS property is not supported or invalid.

The background of the slide is a close-up, low-angle photograph of a computer keyboard. The keys are white and slightly blurred, creating a sense of depth. A white grid pattern is overlaid on the entire image, consisting of thin, intersecting lines that form a mesh. The overall color palette is a mix of light blues, greys, and whites, giving it a clean, technical appearance.

Configuration and Management

4 Log Service Configuration

4.1 Overview

The configuration of Singleton properties specific to the Log Service is described in this section. These properties appear in the Administration Manager, a graphical user interface (GUI) based administration tool included with the OpenFusion Graphical Tools.

The Administration Manager can be used to set the Singleton properties. These properties can also be set programatically, generally as described in the service description sections.

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

Common Properties

Instances of some common properties are used by a number of different OpenFusion Log Service' 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 is 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

4.2 LogFactorySingleton Configuration

This section lists the properties for each of the Log Factory Singletons:

- *BasicLogFactorySingleton*
- *NotifyLogFactorySingleton*

Each Singleton must be configured separately but they have identical properties as defined in this section.

CORBA Properties

IOR Name Service Entry

The Naming Service entry for the Singleton.

<i>Property Name</i>	Object.Name
<i>Property Type</i>	FIXED
<i>Data Type</i>	STRING
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

IOR URL

The *IOR URL* property specifies the location of an Interoperable Object Reference (IOR) for the Service, using the Universal Resource Locator (URL) format. This information is used when a client attempts to resolve a reference to the Service. Some examples are:

```
file:/usr/users/openfusion/servers/NameService.ior
http://www.primstech.com/of/servers/NameService.ior
corbaloc::server.primstechnologies.com/NameService
```

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.

<i>Property Name</i>	IOR.URL
<i>Property Type</i>	FIXED
<i>Data Type</i>	URL
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	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.

<i>Property Name</i>	IOR.File
<i>Property Type</i>	FIXED
<i>Data Type</i>	FILE
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

Resolve Name

The ORB Service resolution name used to resolve calls to the Singleton.

<i>Property Name</i>	ResolveName
<i>Property Type</i>	FIXED
<i>Data Type</i>	STRING
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	YES

IOR Name Service

The name of the Naming Service which will be used to resolve the Singleton object.

<i>Property Name</i>	IOR.Server
<i>Property Type</i>	FIXED
<i>Data Type</i>	STRING
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

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*

4.2 LogFactorySingleton Configuration

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 and must be held locally on the machine running the Service.

The write-ahead log may increase performance when persistent events are required, particularly when events are being delivered quickly.

The write-ahead log is enabled when this property is set `TRUE` (checked). By default, the write-ahead log is disabled. It can be safely enabled unless your system is not set up to allow files to be written to the local machine.

<i>Property Name</i>	DB.WAL
<i>Property Type</i>	STATIC
<i>Data Type</i>	BOOLEAN
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	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>/LogService/data
```

where `<INSTALL>` is the OpenFusion installation path. See the *System Guide* for details of the `domains` directory structure.

<i>Property Name</i>	DB.WAL.Dir
<i>Property Type</i>	STATIC
<i>Data Type</i>	DIRECTORY
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	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.

<i>Property Name</i>	DB.WAL.MaxSize
<i>Property Type</i>	STATIC

<i>Data Type</i>	INTEGER
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

Database Plugin Class

The database plugin class used for persistent storage of logs. If a custom log store plugin is used, the class should be specified here.

If this property is left blank, the default OpenFusion database plugin implementations will be used.

<i>Property Name</i>	DB.Plugin
<i>Property Type</i>	STATIC
<i>Data Type</i>	STRING
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

General Properties

Enable Event Queueing

Indicates if a queue should be used before the events have been sent to persistent storage. This property can accelerate the speed of saving events when using slow storage.

<i>Property Name</i>	Queue.Enable
<i>Property Type</i>	STATIC
<i>Data Type</i>	BOOLEAN
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

File Queueing Directory

Indicates the directory where the queue is located.

<i>Property Name</i>	Queue.Dir
<i>Property Type</i>	STATIC

4.2 LogFactorySingleton Configuration

<i>Data Type</i>	DIRECTORY
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

Initial Number of Logs

The specific properties for each Singleton are:

- *Initial Number of Basic Logs*
- *Initial Number of Notify Logs*

This property determines the initial number of logs (of the appropriate type) created when the Service is run for the first time.

<i>Property Name</i>	InitialLogs
<i>Property Type</i>	STATIC
<i>Data Type</i>	INTEGER
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

Maximum Number of Queue Files

Indicates how many files can be used to save the queued events where the *maximum number of queue file* multiplied by the *maximum queue file size* is the maximum size that the queue can use.

<i>Property Name</i>	Queue.MaxFiles
<i>Property Type</i>	STATIC
<i>Data Type</i>	INTEGER
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

Maximum Queue File Size

The maximum size of the file which the queued events are saved to.

<i>Property Name</i>	Queue.MaxFileSize
<i>Property Type</i>	STATIC

<i>Data Type</i>	INTEGER
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

Max Records Returned From Query

Sets the maximum number of records that will be in the RecordList returned by a query operation. The default value is 100.

If the query returns more records than the maximum set for the RecordList, the additional records can be obtained using the Iterator.

<i>Property Name</i>	MaxRecordsReturnedFromQuery
<i>Property Type</i>	STATIC
<i>Data Type</i>	INTEGER
<i>Accessibility</i>	READ-WRITE
<i>Mandatory</i>	NO

Order query results

When set to `TRUE` (checked), the records returned by a query will be ordered by time, earliest records first. By default, this property is `FALSE` and query results are unordered.

Only the records returned in the RecordList will be ordered in this way. Additional records returned by the Iterator may or may not be ordered. (The *Max records returned from query* property, above, sets how many records will be returned in the RecordList.)



Ordering is performed in memory so if a large number of records are returned, a correspondingly large amount of memory will be used.

<i>Property Name</i>	OrderQueryResults
<i>Property Type</i>	STATIC
<i>Data Type</i>	BOOLEAN
<i>Accessibility</i>	READ-WRITE
<i>Mandatory</i>	OPTIONAL

4.3 ProcessSingleton Configuration

IOR Name Service Entry

The Naming Service entry for the Singleton.

<i>Property Name</i>	Object.Name
<i>Property Type</i>	FIXED
<i>Data Type</i>	STRING
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	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.prismsystems.com/openfusion/ProcessSingleton.ior
```

<i>Property Name</i>	IOR.URL
<i>Property Type</i>	FIXED
<i>Data Type</i>	URL
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	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.

<i>Property Name</i>	IOR.File
<i>Property Type</i>	FIXED
<i>Data Type</i>	FILE
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

IOR Name Service

The name of the Naming Service which will be used to resolve the Singleton object.

<i>Property Name</i>	IOR.Server
<i>Property Type</i>	FIXED
<i>Data Type</i>	STRING
<i>Accessibility</i>	READ/WRITE
<i>Mandatory</i>	NO

4.3 ProcessSingleton Configuration

5 Log Service Manager

5.1 Overview

The following tasks can be carried out from the Log Service Manager:

- Create Log objects.
- Configure Log objects.
- Run queries that return specified Log records.
- View, edit, and delete Log records.
- Add and remove attributes belonging to a Log record.

5.2 Using the Log Service Manager

To start the Log Service Manager, right-click either the (*BasicLogFactorySingleton*, or *NotifyLogFactorySingleton* in the *Object Hierarchy* and select **Log Manager** from the pop-up menu. See 2.3, *Administration Manager*, on page 14 for details.

The Log Service Manager is shown in *Figure 7*.

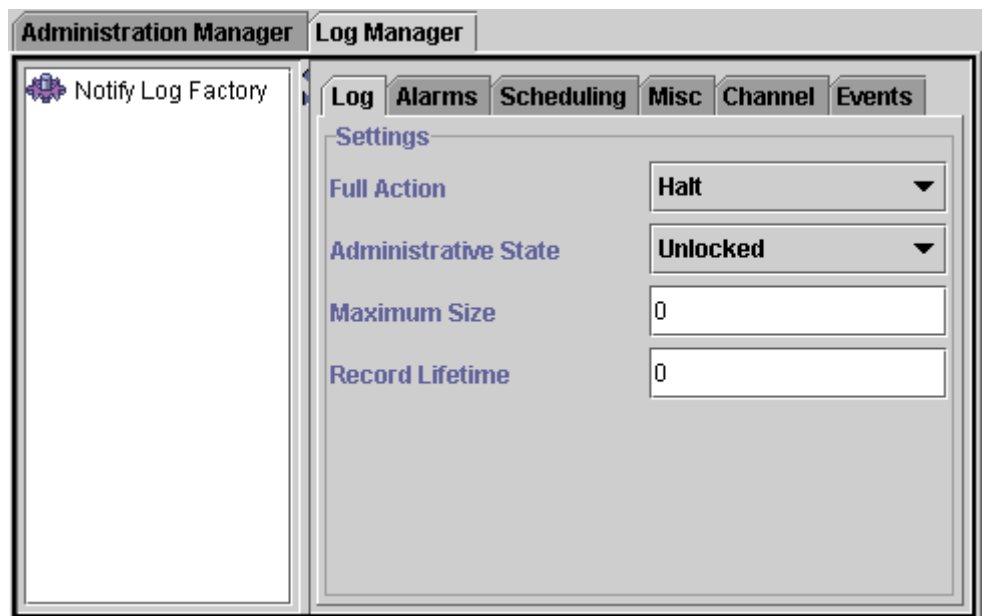




Figure 7 Log Service Manager

Nodes in the Log Service Manager are identified by different icons. These icons are shown in *Table 4*.

Table 4 Log Service Object Icons

Icon	Node
	<p><i>Log Factory</i></p> <p>The root node of the Log Service hierarchy. The description next to this icon indicates which Log Factory is being browsed.</p>
	<p><i>Log</i></p> <p>A representation of a Log object.</p>

Log Object Settings

The Log Manager has six tabbed panels with settings configuring Log objects and capturing Log Service events. Not all Log Managers have all six panels. These panels are described in the following sections. All settings must be correctly configured before a new Log object can be created (see *Create a New Log* on page 93).

Log

The *Log* panel is used by all Log Factories. The following properties must be set on the *Log* panel:

Full Action

The action to be taken when a log is full. Available actions are:

- *Halt* - No further log records will be written to the persistent log record store. The administrative state of the log will be changed to Locked.
- *Wrap* - The oldest logs are deleted to make space for the new ones.

Administrative State

This can be either:

- *Locked* - The Log Service will not write any log records to the persistent store.
- *Unlocked* - The Log Service can write log records to the persistent store.

Maximum Size

The maximum size of the log, in bytes, kilobytes, or Megabytes.

To specify a size in kilobytes, append `kb` to the number. To specify a size in Megabytes, append `Mb` to the number. For example: `100Kb`, `100Mb`. If no suffix is appended, a value in bytes is assumed. Note that case is not significant, so `KB`, `kb`, and `Kb` are all valid suffixes.

A value of `zero` means that there is no maximum limit to the log size.

Regardless of what units are specified for the property, they will be converted to bytes and displayed as such.

Record Lifetime

The length of time, in seconds, minutes, hours, or days, that a log record will exist in the log store before it is automatically deleted.

To specify a lifetime in minutes, append `m` to the number. To specify a lifetime in hours, append `h` to the number. To specify a lifetime in days, append `d` to the number. If no suffix is appended, a value in seconds is assumed. Note that case is not significant, so `D` and `d` are both valid suffixes.

A value of `zero` means that log records never expire.

Regardless of what units are specified for the property, they will be converted to seconds and displayed as such.

Alarms

The *Alarms* panel is used by all Log Factories.

The *Alarms* panel allows unique threshold values to be added to logs. An alarm threshold must be an integer value between `1` and `100`. If several thresholds are entered, each must be a different value.

To add a threshold, enter the value in the *Enter value* field and click the **Add** button.

To remove a threshold, select the threshold in the list and click the **Remove** button.

To set the range of thresholds as the defaults for all logs, click the **Set As Defaults** button. All values are stored together as a set of defaults.

To re-use previously set default values, click the **Use Defaults** button. Note that it is possible to restore previous defaults and then add or delete thresholds in this instance without altering the default set (although changes could be saved as new defaults).

Scheduling

The *Scheduling* panel is used by all Log Factories. These settings allow a schedule to be set up for starting and stopping logging activity.

Current setting

The current setting can be:

- *Default*
- *Custom*

If the *Default* setting is selected, the default *Start* time of `Immediately` and *Stop* time of `Never` are used, and these settings cannot be changed. To manually configure the schedule, select the *Custom* setting.

Start and Stop

If the *Custom* setting is selected, the *Start* and *Stop* times must be manually configured.

Enter the required times and click the **Apply** button.

Week Mask

For fine-grained scheduling, a week mask can be set up. This allows different start and stop times to be set for different days of the week.

Miscellaneous

These settings are used only by the Notify Log Factory.

Forwarding State

Select whether log forwarding is *On* or *Off*.

- *On*: The log will forward incoming events from all suppliers to all consumers currently connected.
- *Off*: The log will not forward incoming events from suppliers to any consumers.

Filter Constraints

Filter constraints are only available for Notify and TypedNotify Logs.

To enter a filter constraint, type the filter expression into the *Value* field and click the **Add** button.

To delete a filter constraint, select the entry in the list and click the **Remove** button.

To edit a filter constraint, select the constraint in the list so that it is displayed in the *Value* field. Edit the constraint value as required, then click the **Replace** button. The new constraint replaces the previous one in the list. If **Add** is clicked instead of **Replace**, the modified constraint is added to the list as a new constraint and the existing constraint will remain in the list.

Filter constraints added here will be applied to any future logs that are created with the Log Service Manager. The filter will not be applied to any logs created programmatically or to any logs which were created in the Log Service Manager *before* the filter was added.

To set a filter constraint on an existing log, select the log in the tree view. Select the *Settings* tab and the *Misc* tab, and add the filter constraint as described above. This filter will apply to the selected log only.

Channel

These settings are used only by the Notify Log Factory. This panel allows Quality of Service (QoS) and Administrative settings to be set for the log.

The following properties can be set:

QoS Settings

- AcknowledgeLevel
- AcknowledgeMode
- ConnectionReliability
- DeadLetterDrop
- DiscardPolicy
- DisconnectCallback
- EventReliability
- EventIDSupported
- LazyAcknowledgeInterval
- MaximumBatchSize
- MaxEventsPerConsumer
- MaxInactivityInterval
- MaxReconnectEvents

5.2 Using the Log Service Manager

- MultipleReceiversSupported
- OrderPolicy
- PacingInterval
- Priority
- ReconnectInterval
- StartTimeSupported
- StopTimeSupported
- ThreadIdleTime
- ThreadPoolSize
- Timeout

Administrative Properties

- MaxQueueLength
- MaxConsumers
- MaxSuppliers
- RejectNewEvents

Events

This panel is used only by the Notify Log Factory and displays information about the events stored in the Log.

The following details are displayed:

- *Event*
- *Time*
- *Log*
- *Details*

The types of events generated by Log Factories and Logs are as follows:

- *StateChange*: Generated when a Log's admin or operational state is changed.
- *ObjectCreation*: Generated when a Log is created.
- *ObjectDeletion*: Generated when a Log is deleted.
- *ThresholdAlarm*: Generated when a Log threshold has been exceeded.

- *AttributeValueChange*: Generated when any of the following Log attributes are changed:
 - Capacity alarm threshold
 - Log full action
 - Max Log size
 - Start time
 - Stop time
 - Week mask
 - Filter
 - Max record life
 - Quality of Service (QoS)

Create a New Log

To create a new Log object, right-click on the Log Factory root node and select **Create** from the pop-up menu.

This will create a new Log object as a child of the root node, but only if all properties have been properly configured. (See *Log Object Settings* on page 88.)

The **Create With ID** option does the same thing as the **Create** option but prompts for a Log ID. A valid unique ID must be entered for the Log object to be created.

The Log node in the Log Factory hierarchy displays the Log's unique number and description. To view information about a log, select the log in the hierarchy. Details are displayed in the right-hand pane, as shown in *Figure 8*.

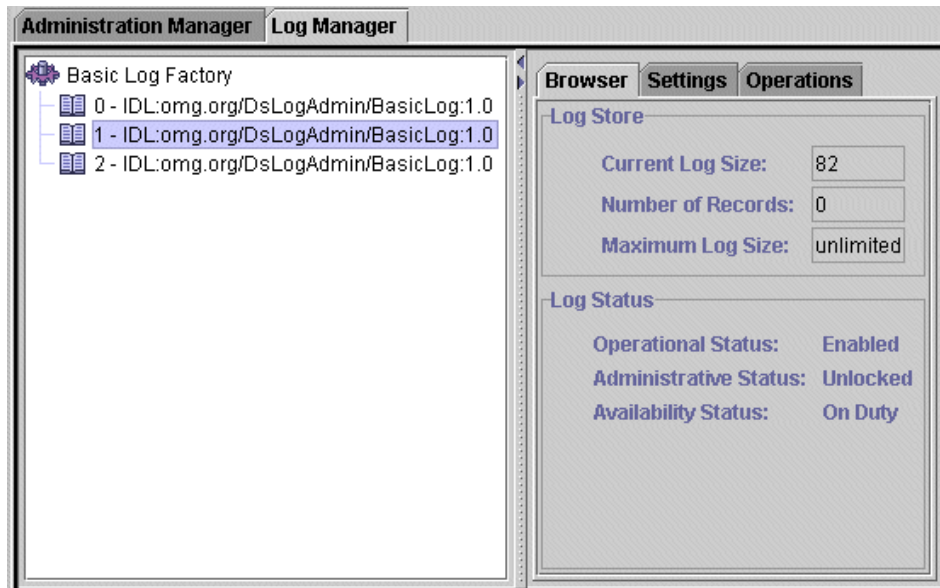


Figure 8 Viewing Log Details

Copy Logs

To make a copy of the Log object, right-click on the Log object in the hierarchy and select **Copy** from the pop-up menu.

The new Log object is added to the Log Factory hierarchy and retains all the configuration settings of the original.

The **Copy with ID** option does the same thing as the **Copy** option but prompts for a Log ID. A valid unique ID must be entered for the copy of the Log object to be created. The ID cannot be the same as the original Log.

Destroy Logs

To remove a Log object from the Log Factory hierarchy and from the Log Service, right-click the Log object and select **Destroy** from the pop-up menu.

Browser

The *Browser* panel displays information about the current status of the Log Service. The following details are displayed:

- *Current Log Size.*
- *Number of Records.*

- *Maximum Log Size*.
- The percentage of the allocated Log *Capacity* currently in use (also summarised in a pie chart).
- *Operational Status* (Enabled or Disabled).
- *Administrative Status* (Locked or Unlocked).
- *Availability Status*.
- Number of *Connected Consumers*.
- Number of *Connected Suppliers*.

For details of these properties, see *Log Object Settings* on page 88.

Settings

The *Settings* panel allows Log object properties to be changed. The panel contains properties on five tabbed panels, corresponding to the properties that can be set when a new Log is created, as follows:

- *Log* (page 88).
- *Alarms* (page 89).
- *Scheduling* (page 90).
- *Misc* (page 90).
- *Channel* (page 91).

Changes to a property in a text box are **not** applied to the Log object until focus is moved to another property *in the same panel*. Navigating to another panel without doing this will cause the change to be lost. (This does not apply to properties which are set with a check box or drop-down list.)

Operations

The *Operations* panel is used to retrieve and display selected Log records.

Five different retrieval actions can be performed on the *Operations* panel. Select which action to perform from the *Operation* drop-down list:

- *Query*
- *Retrieve*
- *Match*
- *Delete*

- *Set Attributes*

Query

The *Query* operation selects and displays records that match a query string.

Step 1: Use the *Grammar* drop-down list to select the query language, `EXTENDED_TCL` (ETCL). Refer to the Notification Service documentation for details of ETCL.

The database plugin used for Log persistence may restrict the choice of grammar.

Step 2: Enter the query string in the *Constraint* field.

Step 3: Click the **execute** button to run the query. Each returned record is displayed in the *Record Details* table.

Step 4: Select a record to display the record's attributes in the *Attribute Details* table.

Records can be individually deleted by selecting them and using the **Delete** button. Deleted records are removed from the Log Service.

Attributes can be added or deleted from the record using the **Add** and **Remove** buttons.

Retrieve

The *Retrieve* operation selects and displays a specified number of log records.

- Enter a positive number in the *Records to return* field to return records starting at the specified time.
- Enter a negative number in the *Records to return* field to return records immediately prior to the specified time.

For example, with `30 March 2001 12:00` as the date:

- *Records to return* - 50 will retrieve the first 50 records logged after 12:00 on 30 March 2001.
- *Records to return* -50 will retrieve the last 50 records logged before 12:00 on 30 March 2001.

Each returned record is displayed in the *Record Details* table. Records can be individually deleted by selecting them and using the **Delete** button. Deleted records are removed from the Log Service.

Select a record to display the record's attributes in the *Attribute Details* table. Attributes can be added or deleted from the record using the **Add** and **Remove** buttons.

Match

The *Match* operation is similar to the *Query* operation but instead of returning a list of Log messages, it returns a count of messages.

Delete

The *Delete* operation deletes all records that match a query string.

Step 1: Use the *Grammar* drop-down list to select the query language, `EXTENDED_TCL` (ETCL). Refer to the Notification Service documentation for details of `ETCL`.

The database plugin used for Log persistence may restrict the choice of grammar.

Step 2: Enter the query string in the *Constraint* field.

Step 3: Click the **execute** button to delete all matching Log records.

A pop-up message box will report the number of records deleted by the query.



Warning: deleted Log records **can not** be recovered.

Set Attributes

The *Set Attributes* operation allows a batch update of attributes for all Log records which match a query string.

Step 1: Click the **Add** button to add an attribute to the *Attribute Details* table.

Step 2: Enter a *Name*, *Value*, and *Type* for the attribute.

Step 3: Repeat steps 1 and 2 to add more attributes to the table.

Step 4: Use the *Grammar* drop-down list to select the query language, `EXTENDED_TCL` (ETCL). Refer to the Notification Service documentation for details of `ETCL`.

The database plugin used for Log persistence may restrict the choice of grammar.

Step 5: Enter the query string in the *Constraint* field.

Step 6: Click the **execute** button to run the query.

5.2 Using the Log Service Manager

Each Log record retrieved by the query is automatically updated to include the attributes added to the *Attribute Details* table.

A pop-up message box will report the number of records updated by the query.



Warning: this change **can not** be undone.

A close-up, low-angle view of a computer keyboard, showing several keys in detail. The keys are white and the keyboard is set against a dark background. A white grid pattern is overlaid on the entire image, creating a sense of depth and structure. The word "Appendices" is written in a bold, dark blue font across the upper portion of the image.

Appendices

Appendix A

Locating a Log with the Naming Service

Distributed CORBA objects can be readily located using the Naming Service. The Naming Service associates a name with an object. This name can then be used to efficiently locate the object. The association between an object and a name is called a *name binding*.

The Naming Service stores name bindings in a hierarchical arrangement called a *naming context*. As well as organising name bindings, contexts are used to manage the bindings, including their creation and destruction. The top-most context is called the *root context* and may contain *child* contexts, as well as name bindings. The entire arrangement of root context, child contexts, and name bindings is called a *naming graph*.

Creating a Name Binding

In order to create a name binding, a reference to the root context (or child context) where it will be stored must be obtained. The name itself is stored in a *NameComponent* object, which is then added to the context. The following code, taken from *Locating the Log* on page 24, shows the basic approach to binding a log object to a name.



The *OpenFusion Naming Service Guide* provides complete details on using contexts and name bindings.

The object to be bound and an `int` holder are declared. The initialisation of the `orb` object is not shown. The name binding is then created as described in the steps given below.

```
org.omg.CORBA.ORB orb = null;
BasicLog log = null;
org.omg.CORBA.IntHolder logId = null;

// bind the log object to a name with Naming Service
org.omg.CORBA.Object namingService = null;
NamingContextExt rootContext = null;
NameComponent basicLog[];

namingService = orb.resolve_initial_references ("NameService");
rootContext = NamingContextExtHelper.narrow(namingService);
basicLog = new NameComponent [1];

basicLog[0] = new NameComponent ("BasicLog", "log");
rootContext.rebind(basicLog, log);
```

```
System.out.println("BasicLog created.");
```

- Step 1:** Declare a CORBA object which will be resolved to the Naming Service, a root context object (of type *NamingContextExt*), and an array of type *NameComponent*. The *NameComponent* array is used to contain the name which will be used to identify the log object.
- Step 2:** Resolve the Naming Service object using the standard CORBA *resolve_initial_references()* method, then narrow the Naming Service object to a root context using *NamingContextExtHelper.narrow()*. Initialise the *NameComponent* array, *basicLog*, to hold the name identifying the log.
- Step 3:** Assign a new name component to *basicLog*, using two *String* parameters defining its *id* and *type*. The *id* and *type* can be any desired string, or even contain *NULL* values, provided *NULL* is not used for both *id* and *type*. Also, the *id-type* combination must be unique within the context. Finally, bind the log object to the name component using root context's *bind()* or *rebind()* methods. The *bind* method will throw an exception if the name component is already bound to an object; *rebind* will replace an existing binding or create a new binding if one does not exist. An object can be bound to more than one name component, but one name component can not be bound to more than one object.

A name binding can be removed using the root context's *unbind()* method. If a context is empty (i.e., does not contain any name bindings or child contexts), it can be destroyed using the context's *destroy()* method.

i Removing a name binding does **not** destroy the object that the name is bound to.

The log object can now be located, from clients located anywhere in the system, using the context's *resolve()* method described below.

Obtaining the Object with Resolve

Clients can now obtain a reference to the log by using the *resolve()* method. This is referred to as *resolving* the object.

An object can be resolved by simply passing a *NameComponent* object (containing the log's *id* and *kind* values) to the context's *resolve()* method, then narrowing the object to the *BasicLog* type (*resolve()* returns an *Object* type).

```
// obtain reference to the orb
orb = ORBAdapter.init(args);

// obtain reference to log using Naming Service
obj = orb.resolve_initial_references("NameService");
rootContext = NamingContextExtHelper.narrow(obj);
```



```
basicLog = new NameComponent [1];  
basicLog[0] = new NameComponent ("BasicLog", "log");  
obj = rootContext.resolve(basicLog);  
log = BasicLogHelper.narrow(obj);
```


Appendix B

Using with JacORB

If the OpenFusion Log Service's NotifyLog is used with the NotifyLog with *jacORB*, then *jacORB*'s *Compact Type Codes property* must be turned off by setting it to 0 (zero). The Compact Type Codes property is located in *jacORB*'s *jacorb.properties* file. The default location of the *jacorb.properties* file is the *install/classes* directory.

The property should be changed from:

```
jacorb.compactTypecodes=2
```

to:

```
jacorb.compactTypecodes=0
```


A close-up, low-angle view of a computer keyboard, showing several keys in detail. The keys are white and set against a dark background. A white grid pattern is overlaid on the entire image, creating a sense of depth and perspective. The word "Index" is printed in a dark blue, sans-serif font in the upper right quadrant of the image.

Index

Index

A

AcknowledgeLevel (QoS property)	91	unlocked	14, 24
AcknowledgeMode (QoS property)	91	Administrative State (log setting)	88
add_constraints	54, 55	Alarms	89
add_filter	55, 66	Architecture	6, 8
Administration Manager	5	arg_list	12
Administrative Properties	92	attr_list	12, 30
Administrative State	13, 14, 23, 88	Attribute Change Events	17
locked	14, 23	AttributeValueChange	13, 14, 17, 93
Set	23, 24	Availability Status	16, 29

B

BasicLog	6, 7, 8, 20, 21, 26	BasicLogFactory	9, 23
Creating	21	BasicLogFactorySingleton	77, 87
Creating a Client	25	Configuration	77
BasicLog (including Interval)	25	Benefits	3
BasicLog (including Write and Query Related)		bind	21, 102
31		Browser	94

C

Callback Operations	37, 44, 46	Configuration	22, 33, 50, 62
Push	71	ConnectionReliability (QoS property)	91
push_structured_event	67	Constraint Expression	29, 54
Capacity	12	Contacts	viii
Channel	11, 91	Context	25
child context	101	Conventions	vii
Cleanup	48, 69	copy	6, 13
Client		Copy Logs	94
Connecting to an Event Channel	40	copy_with_id	13
Event Consumer	46	CORBA Properties	78
Event Supplier	40	CosEvent	37
Terminating	42	Create	22, 32, 34
Clients	20	create_filter	54
ClientType	59	Current setting	90
Concepts	6	Current setting (Log Scheduling)	90

D

Data, Sending	27	Database Plugin Class (property)	81
-------------------------	----	--	----

Basic Log Factory	81	Dependencies (on Other Services).	5
DB.Plugin (property)	81	Dependencies on Other Services.	5
DB.WAL (property).	80	Destroy	6, 13
DB.WAL.Dir (property)	80	destroy	35, 42, 54, 56, 102
DB.WAL.MaxSize (property)	80	Destroy Logs.	94
DeadLetterDrop (QoS property).	91	DiscardPolicy (QoS property).	91
Default		disconnect_push_supplier	37, 39
Values.	22, 46	DisconnectCallback (QoS property)	91
Default Grammar	54	Duration	15, 26
Delete.	97		

E

Enable Event Queueing (property)	81	EventLogFactory.	10, 34
Enable Write Ahead Log (property)	79	EventReliability (QoS property)	91
Event		Events.	92
Filtering	66	Creating and Sending	61
Event Channel	6, 11, 37, 40, 51	Filtering	53
Event Header	60	Forwarding	17, 46
Event Supplier	37	Receiving.	46, 67
EventIDSupported (QoS property).	91	Sending	41
EventLog	7, 32, 38, 48	Working with	32
Creating	32	EventType.	54, 66
Creating a Consumer Client	43	Exceptions 25, 31, 36, 43, 48, 56, 63, 69, 72,	
Creating a Supplier Client.	57	73	
Duration	36	EXTENDED_TCL	
Time Values	44	SeeExtended Trader Constraint Language	

F

Factories		Set Record Filter	55, 66
Log	6, 9	filter	
Factory Id.	14	get	55
Features.	18	remove	56
Comparison	18	Filter Factory.	66
File Queueing directory (property)	81	filterable_body	60
Filter	53	filterable_data	68
add constraints	54, 55	Filtering.	7, 13, 17, 52, 53, 66
Add Event Filter	55, 66	Fixed Header	55, 60
Constraints	90	Forwarding	7, 14
Create	54	Forwarding State	17, 46, 90
Creation	54	Full Action.	88
Location	53	Full Action (property)	88
Removal and Destruction	56		

G

General Properties	81	ProcessingErrorAlarm	13
Generated Events	6, 7, 12, 70, 72	StateChange	13
Monitoring	70	ThresholdAlarm	12, 15
Receiving	71	get_availability_status	51
generated events		get_filter	55
AttributeValueChange	17	Grammar	29, 54
ObjectCreation	12	Gregorgian Calander	23, 30, 35, 47
ObjectDeletion	12		

H

halt	15, 33, 51	Header	55, 60
----------------	------------	------------------	--------

I

id	12, 14, 30	IOR Name Service	85
info	12, 30	IOR Name Service (property)	79, 85
Inheritance		IOR Name Service Entry	84
Hierarchy	7, 10	IOR Name Service Entry (property)	78, 84
Initial Number of Logs (property)	82	IOR URL	84
InitialLogs (property)	82	IOR URL (property)	84
Instrumentation	5	IOR.File (property)	79, 85
interface_id	12	IOR.Server (property)	79
IOR File Name	84	IOR.URL (property)	78, 84
IOR File Name (property)	84		

L

LazyAcknowledgeInterval (QoS property)	91	NotifyLog	6, 7
Lifecycle Operations	13	TypedEventLog	7
Lifetime	16, 89	TypedNotifyLog	7
Locating the Log	24, 36	Types	6, 7, 8, 10
locked	14, 23	Features	18
Log	6, 11, 88	Log Capacity Alarm Threshold	15
BasicLog	6, 7, 8	Log Clients	20
Configuration	22, 33, 50, 62	Log Creation	20, 93
Create	22, 32, 34	Log Creation and Configuration	22, 33, 50
EventLog	7, 32	Log Duration	15, 26
Factories	6, 9	Log Factories	9
Generated Events	6, 7, 12, 70, 72	Log Factory	88
Locating	24, 36	Log Full Actions	15, 34
Management	14	halt	15, 33, 51
Maximum Log Size	14, 22	wrap	15

Log Generated Events	12	Log Service	
Monitoring	70	Configuration	77
Log Management	14	Exceptions	73
Log Networks	11	Manager	87
Log Object	6	Supplemental Info	73
Log Object Settings	88	Using Specific Features	19
Log Record	11, 16, 22, 30, 51	Log Service Manager	
filtering	17	Using	87
Lifetime	16	Log Types	6
retrieve	29, 30, 38, 46, 47, 68	LogFactorySingleton Configuration	77
Log Records		LogMgr	9
Retrieving	47, 68	LogRecord	11
Log Scheduling	15		

M

Main Filter Components	53	Maximum queue file size (property)	82
Management	14	Maximum Size	89
Additional Operations	16	Maximum Size (log setting)	89
Management Operations		MaximumBatchSize (QoS property)	91
Common	14	MaxInactivityInterval (QoS property)	91
Match	97	MaxQueueLength (admin property)	92
Max records returned from query (property)	83	MaxReconnectEvents (QoS property)	91
MaxConsumers (admin property)	92	MaxRecordsReturnedFromQuery (property)	83
MaxEventsPerConsumer (QoS property)	91	MaxSuppliers (admin property)	92
Maximum Log Size	14, 22	MultipleReceiversSupported (QoS property)	92
Maximum number of queue file (property)	82		

N

Name Binding	101	Root Context	101
Creating	101	Notification Service	
NameComponent	27, 36	Introduction	3
Naming Graph	101	Service Dependencies	5
Naming Service	26, 70, 101	NotifyLog	6, 7, 20, 49, 57, 58, 64, 69, 72
bind	21, 102	Creating	49
Child Context	101	Creating a Consumer Client	64
Context	25	Creating a Supplier Client	37
Name Binding	101	Structured Events	48
Naming Graph	101	Time Values	64, 70
Resolve	102	NotifyLogFactory	10, 33
resolve	25	NotifyLogFactorySingleton	77, 87

O

Object.Name (property)	78, 84	operation_name	12
ObjectAdapter	37, 60	Operations	95
ObjectCreation	12, 92	ORB	22
ObjectDeletion	12, 92	Orb Abstraction Layer	22
obtain_push_supplier	46	orb abstraction layer	37, 60
OMG		Orb Adaptor	21, 26, 32, 38, 64, 70
Standard Features	4	ORBAdapter	
OpenFusion		Also see ObjectAdapter	
Enhancements	4	Order query results (property)	83
Extensions	20	OrderPolicy (QoS property)	92
Features	4	OrderQueryResults (property)	83
Orb Adaptor	21, 26, 32, 38, 64, 70	Organisation	vi

P

PacingInterval (QoS property)	92	Proxy	
Persistence Properties	79	See Proxies	
Priority (QoS property)	92	Proxy Filter, Setting a	66
ProcessingErrorAlarm	13	Pull Model	40, 46
ProcessSingleton Configuration	84	Push	40, 41, 44, 46, 71
Properties		Push Model	40, 46
Common	77	push_structured_event	67
General	81	PushSupplierOperations	37, 44
Proxies	40		

Q

QoS		Quality of Service	16, 17, 18, 59
See Quality of Service		QoSFlush	17, 33
QoS Settings	91	QoSNone	17, 33
Log Factories	91	QoSReliability	18, 33
QoSFlush	17, 33	Query	4, 25, 29, 30, 96
QoSNone	17, 33	Querying the Log Records	29
QoSReliability	18, 33		

R

Read and Write Settings	14	See Log Record	
Read-Only Settings	14	record	
Receiving Events	46, 67	See Log Record	
Receiving Generated Events	71	Record Lifetime	89
ReconnectInterval (QoS property)	92	Record Lifetime (log setting)	89
Record		Registering with the Naming Service	56

RejectNewEvents (admin property)	92	resolve	25
remainder_of_body	60, 61, 67	ResolveName (property).	79
remove_filter	56	Retrieve	29, 30, 38, 46, 47, 68, 96
Repository Id	12	Retrieving Log Records	47, 68
Resolve	102	root context.	101

S

Scheduling	15, 90	SNMP	5
Sending Data	27	Start and Stop.	90
Sending Events	41	Start and Stop (log scheduling)	90
Set Attributes	97	Starting Log Service Manager	87
set_administrative_state	23, 24	StartTimeSupported (QoS property).	92
set_filter	55, 66	StateChange.	13, 92
set_interval.	22, 23, 29	StopTimeSupported (QoS property).	92
set_week_mask.	29, 35	Structured Event.	7, 54, 61
Setting, Current	90	header	55, 60
Settings	95	Structured Events	48, 60
Log Object	88	Structured Push Supplier	59
Singletons		StructuredProxyPushSupplier.	65
BasicLogFactorySingleton	77		

T

ThreadIdleTime (QoS property)	92	TypedEventLogFactory.	10
ThreadPoolSize (QoS property).	92	TypedLogRecord	11, 12
ThresholdAlarm	12, 15, 92	arg_list.	12
Time	12, 23, 29, 30	interface_id	12
TimeInterval	23, 35	operation_name	12
Timeout (QoS property)	92	TypedNotifyLog.	7
Timer	26, 38, 58	TypedNotifyLogFactory.	10
TimerTask	26, 38, 58	Types	
TypedEventLog	7	Log	6, 7, 8, 10

U

unlocked.	14, 24	Using the Log Service Manager	87
-------------------	--------	---	----

V

Values		Variable Header	60
Default	22, 46		

W

Week Mask	35, 90	Write Ahead Log	79
Week Mask (Log Scheduling)	90	Write Ahead Log Directory (property)	80
WeekMask	35	Write Ahead Log Maximum Size (property)	80
wrap	15	Write Data	27, 29
Write	27, 29	write_records	29

