ORBacus Notify

Version 2.0.0

CHAPTER 1 Introduction

1.1 What is ORBacus Notify?

ORBACUS NOTIFY is an implementation of the Object Management Group (OMG) Notification Service specification [1]. It is fully backwards compatible with the OMG Event Service specification [2], providing a smooth migration path for applications that use an event service.

Some highlights of ORBACUS NOTIFY are:

- Written in C++ for maximum performance
- Multi-threaded architecture
- Event filtering
- Any, structured, and sequence event types
- Push and pull suppliers and consumers
- Quality of Service (QoS) parameters to control event queueing and event lifetime
- Persistent and best effort event and channel reliability QoS parameters
- Subscription sharing between channels and clients

ORBACUS NOTIFY also features the ORBACUS NOTIFY Console, a graphical user interface, which is written in Java for maximum portability. The user interface supports the maintenance of:

- Event channels
- Supplier and consumer admins
- Proxy consumers and suppliers
- All QoS parameters
- Filters and constraint expressions
- Event subscription and offer information

1.2 About this Document

The ORBACUS NOTIFY manual provides a brief overview of Event and Notification Service concepts. However, this document is not a substitute for the OMG Event Service and Notification Service specifications. Please consult [1] and [2] for a detailed description of these services.

The manual also includes a discussion of configuration issues, an introduction to application development with examples in C++ and Java, and detailed descriptions of the ORBACUS NOTIFY Console and proprietary ORBACUS NOTIFY features.

1.3 Getting Help

Should you need any assistance with ORBacus Notify, please visit our Support Frequently Asked Questions (FAQ) list at `http://www.orbacus.com/faq/support.html`.

CHAPTER 2

Configuration and Startup

2.1 ORBacus Notify

2.1.1 Synopsis

Usage

```
notserv [-v,--version] [-h,--help] [-i,--ior] [-d,--dbdir]
```

Options:

`-v` `--version`	Reports the ORBACUS NOTIFY version number.
`-h` `--help`	Displays `notserv` command information.
`-i` `--ior`	Prints IOR on standard output.
`-d` `--dbdir`	Specifies the path to the database directory (e.g., `--dbdir <database directory>`).

Windows NT Native Service

ORBACUS NOTIFY is also available as a native Windows NT service.

```
ntnotservice  [-h,--help] [-i,--install] [-u,--uninstall]
              [-d,--debug]
```

Options:

`-h` `--help`	Displays command line options supported by the server.
`-i` `--install`	Install the service. The service must be started manually.
`-s` `--start-install`	Install the service. The service will be started automatically.
`-u` `--uninstall`	Uninstall the service.
`-d` `--debug`	Run the service in debug mode.

In order to use ORBACUS NOTIFY as a native Windows NT service, it is first necessary to add the `NotificationService` initial reference to the HKEY_LOCAL_MACHINE NT registry key (see "Using the Windows NT Registry" in the ORBACUS manual for more details).

Next the service is installed with:

```
ntnotservice -i
```

This adds the ORBacus Notify entry to the Services dialog in the Control Panel. To start ORBACUS NOTIFY, select the ORBacus Notify entry and press Start. If the service is to be started automatically when the machine is booted, select the ORBacus Notify entry, then click Startup. Next select Startup Type - Automatic, and press OK.

If you want to remove the service, run:

```
ntnotservice -u
```

Note: If the executable for ORBACUS NOTIFY *is moved, it must be uninstalled and re-installed.*

Any trace information provided by the service is placed in the Windows NT Event Viewer with the title NotifyService.

2.1.2 Configuration Properties

In addition to the standard ORBACUS configuration properties, ORBACUS NOTIFY also supports the following properties:

ooc.notification.dbdir

Value: *directory*

Specifies the directory in which ORBACUS NOTIFY stores its databases. This property must be set, either in a configuration file or on the command line, otherwise ORBACUS NOTIFY will not start.

ooc.notification.dispatch_strategy

Value: `threaded`, `thread_pool`

ORBACUS NOTIFY supports two different models for scheduling push and pull requests on clients. The best dispatch model depends on how ORBACUS NOTIFY is to be used.

`threaded`

Each push supplier and pull consumer proxy has a thread invoking requests on the client supplier or consumer. Each proxy transfers or receives events independent of the other. If there is a large number of consumers or suppliers, this can result in a large number of active threads. This model is useful for environments where communication latency varies from client to client and/or the host system can process multiple threads efficiently. On systems where threads are expensive, it may be preferable to use `thread_pool`.

When using the `threaded` dispatch model, pull consumer proxies invoke `pull()` on pull suppliers.

`thread_pool`

All channels share a “pool” of threads that invoke requests on the client supplier or consumer. There is a fixed number of threads dispatching requests on clients, placing an limit on the number of concurrent push/pull requests. This model is useful for environments where it is desirable to place an upper bound on the number of active threads. The number of threads in the pool are controlled by the `dispatch_threads` property.

When using the `thread_pool` dispatch model, pull consumer proxies invoke `try_pull()` on pull suppliers.

ooc.notification.dispatch_threads

Value: *threads* > 0

Specifies the number of threads for the "`thread_pool`" dispatch strategy. The default is 10.

ooc.notification.endpoint

Value: *string*

Specifies the endpoint configuration for the service. Note that this property is only used if the `ooc.orb.oa.endpoint` configuration property (see [4]) is not set.

ooc.notification.events_per_transaction

Value: *events* > 0

Determines the maximum number of events selected per database transaction for transmission to a push consumer. This property reduces total transaction overhead for persistent events. The default value is 100.

ooc.notification.eventqueue

Value: `true,false`

If `true` a central event queue is used. The default value is `false`, that is the central event queue is not used. The central event queue helps isolate suppliers from consumers at the expense of an increased number of transactions. For configurations with few suppliers and consumers, it is recommended to set this to `false`.

ooc.notification.trace.events

Value: *level* >= 0

Controls the level of diagnostic output related to event lifecycles. Set this value to 1 or greater to enable event lifecycle tracing. The default is 0, which produces no output.

ooc.notification.trace.lifecycle

Value: *level* >= 0

Controls the level of diagnostic output related to service object (channel, admin, proxy) lifecycles. Set this value to 1 or greater to enable service object lifecycle tracing. The default is 0, which produces no output.

ooc.notification.trace.queue

Value: *level* >= 0

Controls the level of diagnostic output related to proxy event queue operations. Set this value to 1 or greater to enable proxy event queue tracing. The default is 0, which produces no output.

ooc.notification.trace.retry

Value: *level* >= 0

Controls the level of diagnostic output related to retried event transmissions. Set this value to 1 or greater to enable event retry tracing. The default is 0, which produces no output.

ooc.notification.trace.subscription

Value: *level* >= 0

Controls the level of diagnostic output related to subscription sharing. Set this value to 1 or greater to enable subscription sharing tracing. The default is 0, which produces no output.

ooc.filter.trace.lifecycle

Value: *level* >= 0

Controls the level of diagnostic output related to filter object (forwarding filter, mapping filter, filter factory) lifecycles. Set this value to 1 or greater to enable service object lifecycle tracing. The default is 0, which produces no output.

ooc.database.trace.transactions

Value: *level* >= 0

Controls the level of diagnostic output from the transaction subsystem. Set this value to 1 or greater to enable database transaction tracing. The default value is 0, meaning no transaction tracing.

ooc.database.trace.database

Value: *level* >= 0

Controls the level of diagnostic output related to database activity. Set this value to 1 or greater to enable database activity tracing. The default value is 0, meaning no tracing.

ooc.database.trace.locks

Value: *level* >= 0

Controls the level of diagnostic output related to database locking. Set this value to 1 or greater to enable database lock tracing. The default value is 0, meaning no tracing.

ooc.database.max_retries

Value: *retries* >= 0

The maximum number of retries of a transaction before an abort. When a transaction is aborted it is completely rolled back and a `CORBA::TRANSIENT` exception is raised meaning the client should retry the request later. A value of 0 means unlimited retries. The default value is 0.

ooc.database.max_sleep_time

Value: *time* >= 0

The maximum amount of time to sleep (in seconds) between retries. The time between successive retries grows exponentially until this value is reached, that is 1, 2, 4, 8,... `max_sleep_time`. Set this value to 0 to disable sleeping between retries. The default value is 256.

ooc.database.checkpoint_interval

Value: *interval* >= 0

The interval at which database checkpointing occurs in seconds, in conjunction with `checkpoint_kbyte`. Set this value to 0 to disable checkpointing. The default is 300 seconds.

ooc.database.checkpoint_kbyte

Value: *kbyte* >= 0

The minimum amount of database log data (in kilobytes) that must be present before a checkpoint occurs. Set this value to 0 to create a checkpoint every `checkpoint_interval` seconds. The default is 64 kilobytes.

ooc.database.sync_transactions

Value: `true,false`

Determines whether transactions are truly synchronous. If set to `false`, event throughput will be higher but there is a risk of events being lost if the service crashes. The default is `true`.

ooc.database.max_locks

Value: *locks* > 0

Configures the maximum number of database locks that may be acquired at any time. The default value is 16384. If it is expected that the database will contain a large number of events at any one time, then this value should be increased.

ooc.database.max_transactions

Value: *transactions* > 0

Configures the maximum number of concurrent transactions that may be active at any one time. This value should be set proportional to the number of persistent proxies. Otherwise, if there are many persistent proxies and not enough concurrent transactions are permitted, performance will decrease. The default is 20.

2.1.3 Connecting to the Service

The object key of ORBACUS NOTIFY is `DefaultEventChannelFactory`, which identifies an object of type `CosNotifyChannelAdmin::EventChannelFactory`. The object key can be used when composing URL-style object references. For example, the following URL identifies the notification service running on host `nshost` at port `10000`:

```
corbaloc::nshost:10000/DefaultEventChannelFactory
```

2.2 ORBacus Notify Console

2.2.1 Synopsis

```
java com.ooc.CosNotifyConsole.Main
```

There are no command line options specific to the ORBACUS NOTIFY Console.

2.3 Startup Example

The following is an example for how to start ORBACUS NOTIFY and the ORBACUS NOTIFY Console, using an ORBACUS configuration file. For more information on ORBACUS configuration files, please refer to [4]. Note that it is also possible to use command line parameters instead of configuration files.

Create a file with the following contents, and save it as `/tmp/ob.conf` (Unix) or `C:\temp\ob.conf` (Windows):

```
1 ooc.notification.endpoint=iiop --port <port>
2 ooc.notification.dbdir=<database directory>
3 ooc.orb.service.NotificationService=corbaloc::<host>:<port>/
  DefaultEventChannelFactory[1]
```

1 Specifies the endpoint configuration for the service. Replace `<port>` with an arbitrary, free TCP port (e.g. `10001`).

2 Specifies the path to the service's database directory. Replace `<database directory>` with the directory where the service should create its databases.

3 Provides a reference to the default event channel factory. Replace `<host>` with your system's host name and `<port>` with the TCP port chosen above.

2.3.1 Starting ORBacus Notify

After ORBACUS NOTIFY has been properly built and installed, there will be a `notserv` executable in the installation target directory. For example, on UNIX, assuming the installation path was set to `/usr/local`, the executable is:

```
/usr/local/bin/notserv
```

And on Windows, with the installation path set to `C:\ORBacus`:

```
C:\ORBacus\bin\notserv.exe
```

You can start ORBACUS NOTIFY as follows:

- Specify the configuration file on the command line:

  Unix

  ```
  /usr/local/bin/notserv -ORBconfig /tmp/ob.conf
  ```

  Windows

  ```
  C:\ORBacus\bin\notserv.exe -ORBconfig C:\temp\ob.conf
  ```

- Specify the configuration file with an environment variable:

  Unix

  ```
  ORBACUS_CONFIG=/tmp/ob.conf
  export ORBACUS_CONFIG
  ```

1. Note that for typesetting reasons this configuration option spans two lines, but in your configuration file, only one single line must be used.

```
/usr/local/bin/notserv
```

Windows

```
set ORBACUS_CONFIG=C:\temp\ob.conf
C:\ORBacus\bin\notserv.exe
```

2.3.2 Starting the ORBacus Notify Console

The Java archive `OBNotify.jar` contains the ORBACUS NOTIFY Console. For example, on Unix, assuming the installation path was set to `/usr/local`, the archive can be found at:

```
/usr/local/lib/OBNotify.jar
```

And on Windows, assuming the installation path was set to `C:\ORBacus`:

```
C:\ORBacus\lib\OBNotify.jar
```

Note that the console application also requires `OB.jar`, `OBUtil.jar`, and `OBEvent.jar` from ORBACUS for Java distribution. Assuming these files are in the same directory as `OBNotify.jar`, the console can be started as follows:

Unix

```
CLASSPATH=/usr/local/lib/OB.jar:/usr/local/lib/OBUtil.jar:/usr/
local/lib/OBEvent.jar:/usr/local/lib/OBNotify.jar:$CLASSPATH
```

```
export CLASSPATH
```

```
java com.ooc.CosNotifyConsole.Main -ORBconfig /tmp/ob.conf
```

Windows

```
set CLASSPATH=C:\ORBacus\lib\OB.jar;C:\ORBacus\lib\OBUtil.jar;C:\O
RBacus\lib\OBEvent.jar;C:\ORBacus\lib\OBNotify.jar;%CLASSPATH%
```

```
java com.ooc.CosNotifyConsole.Main -ORBconfig C:\temp\ob.conf
```

Figure 2.1: shows a screenshot of the console right after startup.

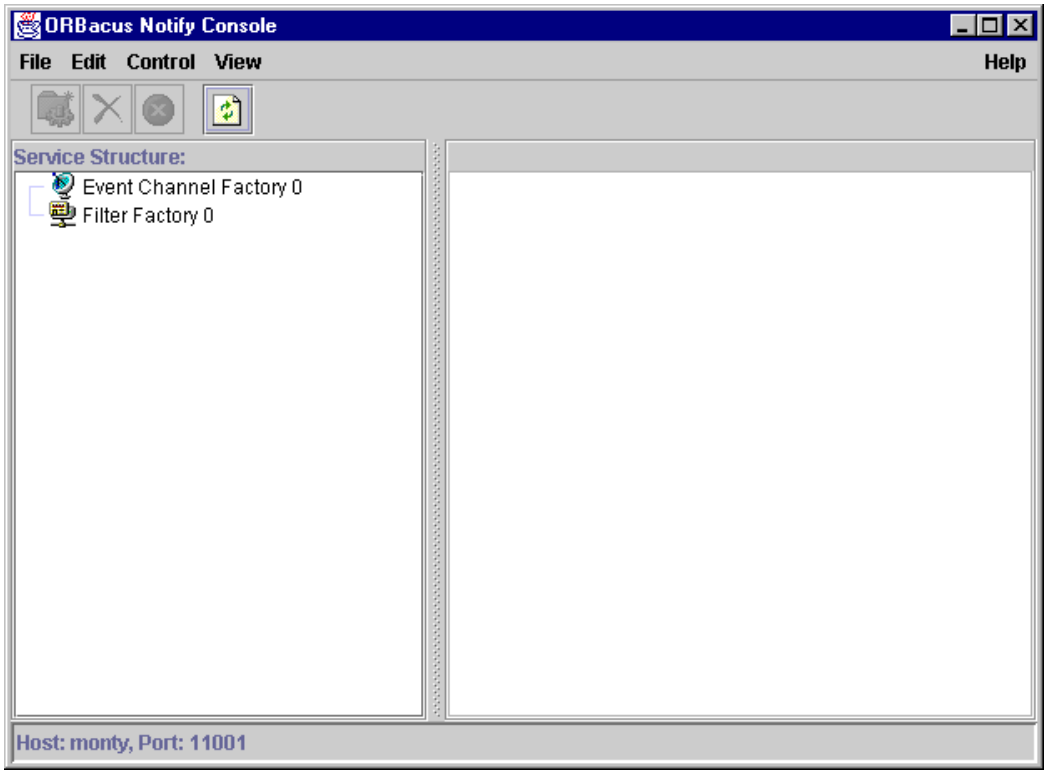

Figure 2.1: Starting the ORBacus Notify Console

CHAPTER 3

Notification Service Concepts

3.1 Overview

In general, CORBA communications are synchronous. A client obtains a reference to a target object, invokes a request on that object, and blocks while waiting for a reply. For some applications the blocking request mechanism is not suitable. An alternative is to implement a distributed callback mechanism allowing applications to make requests on a peer and have that peer notify it asynchronously of the result. This introduces significant complexity since the application must now deal with issues related to peer registration, persistence, managing peer object references, peer unavailability, etc. The effort required to handle such matters may dwarf the application's true purpose.

The OMG Event Service was designed to decouple communications between peer applications, for which the synchronous request model and distributed callback scheme was too restrictive or too complex. The Event Service introduced the concept of the event channel, an entity to which peers could connect to supply and consume events. Clients of the Event

Service are classified as suppliers, consumers, or both depending on how they connect to an event channel. Figure 3.1 illustrates a simplified delivery model:

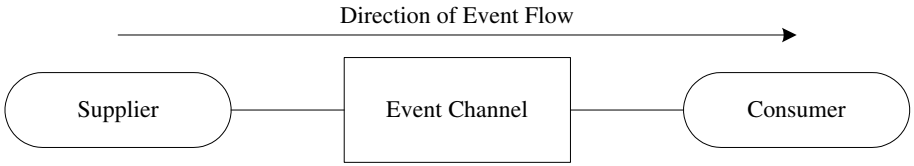

Figure 3.1: Basic Event Service Communications Model

Still, the event service suffers from some serious drawbacks.

Lack of Reliability

The event service makes no guarantees with regards to event delivery or connection persistence. Any level of reliability is vendor specific.

Lack of Structured Events and Event Filtering

In the event service, the structure of events is unknown to the event channel and consumers are forced to handle all events when only a small subset may be of interest. The CPU time necessary to interpret and discard unwanted events may seriously impact consumer performance. This is exacerbated when multiple suppliers are connected to a channel.

Lack of an Event Channel Factory

The event service does not address the issue of channel creation. Instead vendors are forced to define and implement proprietary interfaces for this purpose. As a result event service clients become tied a particular vendor and are not easily ported to other event service implementations.

The OMG has adopted the Notification Service to address these issues while maintaining compatibility with the Event Service. This chapter presents an overview of Event Service and Notification Service concepts.

3.2 The OMG Event Service

This section explains many of the terms and concepts covered by the Event Service. Section 3.3 builds upon this discussion with a presentation of the ideas introduced by the

OMG Notification Service. Refer to specifications [1] and [2] for a complete discussion of the Event Service and Notification Service.

3.2.1 Delivery Models

The mode of event delivery in the Event Service is selected by suppliers and consumers at connection time. The models supported by the event service are discussed next.

Canonical Push Model

In this model, the supplier pushes events to an event channel which in turn pushes events to the consumer (see Figure 3.2). The push supplier is termed *active* since it initiates event

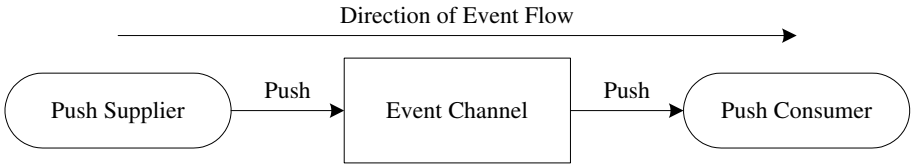

Figure 3.2: Canonical Push Model

delivery with the channel. Conversely the push consumer is *passive* since the channel initiates event delivery.

Canonical Pull Model

In this model, the channel pulls events from the supplier while the consumer pulls events from the channel (see Figure 3.3). A pull supplier is passive since the channel initiates

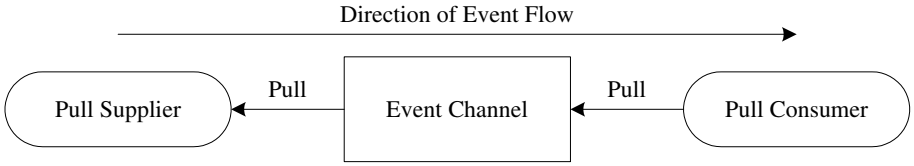

Figure 3.3: Canonical Pull Model

event delivery. A pull consumer initiates event delivery with a channel and is termed active.

Hybrid Push/Pull Model

In the Hybrid Push/Pull model, a push supplier pushes events to an event channel while a pull consumer pulls event from the channel (see Figure 3.4). Both the supplier and con-

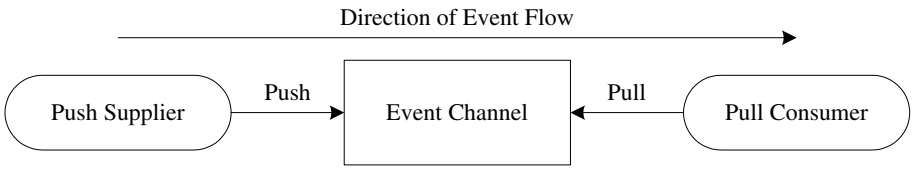

Figure 3.4: Hybrid Push/Pull Model

sumer play active roles in this model.

Hybrid Pull/Push Model

In the Hybrid Pull/Push model, an event channel pulls events from suppliers and pushes them to consumers (see Figure 3.5). The supplier and consumer are both passive in this

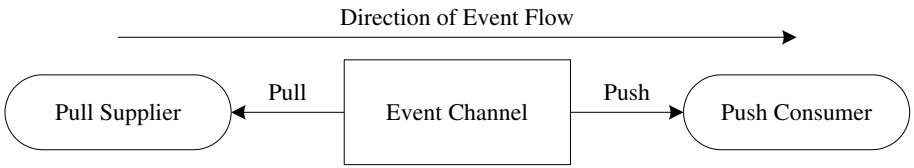

Figure 3.5: Hybrid Pull/Push Model

model.

Combinations of the various models are also supported as illustrated in Figure 3.6.

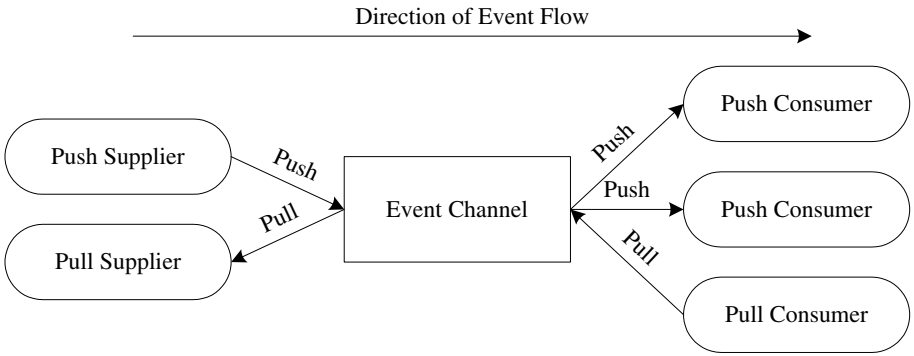

Figure 3.6: Mixed Suppliers and Consumers

3.2.7 Object Management Hierarchy

The relationship between Event Service objects is shown in Figure 3.8.[1] An Event Ser-

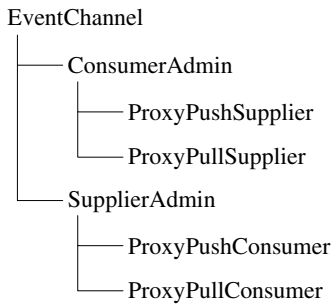

Figure 3.8: Event Service CosEventChannelAdmin Object Management Hierarchy

vice client, ultimately, connects to a proxy object reference so that it may supply or consume events. A set of steps to obtain a proxy object reference are:

1. This diagram is for an untyped event channel. A similar structure exists for typed event channels.

- Obtain an initial reference to an event channel, this is outside the scope of the Event Service specification
- Obtain the appropriate admin object from the channel. Suppliers will want a `CosEventChannelAdmin::SupplierAdmin`, while consumers will want a `CosEventChannelAdmin::ConsumerAdmin`
- Obtain the appropriate proxy from the admin as summarized in Table 3.1

Event Service Client Type	Required Proxy Type
push supplier	`CosEventChannelAdmin::ProxyPushConsumer`
pull supplier	`CosEventChannelAdmin::ProxyPullConsumer`
push consumer	`CosEventChannelAdmin::ProxyPushSupplier`
pull consumer	`CosEventChannelAdmin::ProxyPullSupplier`

Table 3.1: Proxy Selection

- Connect to the proxy

Note: Alternatively, Event Service clients may obtain an object reference (from a naming service, for example) to any of the Event Service objects and then obtain and connect to the proxy.

The proxy, depending on its type, has methods which support the push and pull of events by suppliers and consumers.

3.2.9 Event Delivery

Untyped event delivery in the event service is via a `CORBA::Any`. That is, the event data is unknown to the channel. The proxy interfaces require suppliers to insert event data into a `CORBA::Any` before the event is pushed on or pulled by the channel. Similarly for consumers, all pulled and pushed events are contained within a `CORBA::Any`. Consumers must first extract the event before deciding whether to process or discard it.

3.3 The OMG Notification Service

Much of the previous discussion on the OMG Event Service applies equally to the OMG Notification Service. The Notification Service was designed to be backward-compatible

with the Event Service and it reuses and/or derives from equivalent Event Service IDL interfaces.

3.3.1 Delivery Models

The Notification Service supports the same delivery models as the Event Service, described in Section 3.2.1.

3.3.2 Object Management Hierarchy

The relationship between Notification Service objects is illustrated in Figure 3.3. Note the

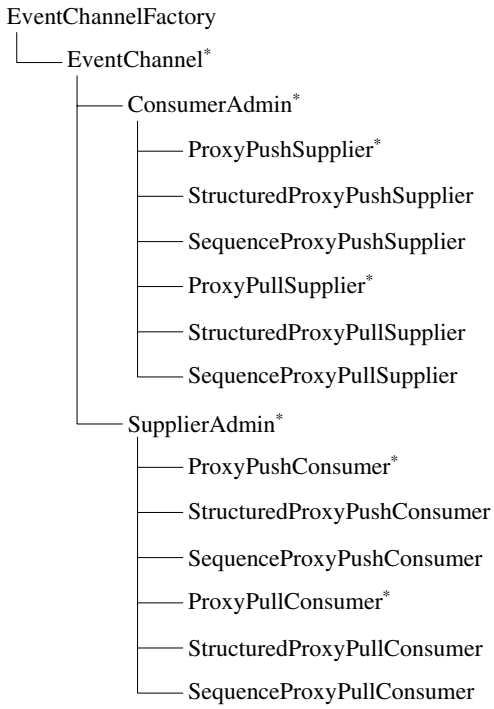

Figure 3.3: Notification Service CosNotifyChannelAdmin Object Management Hierarchy

objects marked with (*); these are Notification Service equivalents of the Event Service

counterparts. Also note the interfaces added by the Notification Service. The `CosNotifyChannelAdmin::EventChannelFactory` addresses the lack of factory issue in the Event Service, while several proxy interfaces have been added to support structured event delivery.

3.3.4 Event Delivery

The Notification Service supports the delivery of events in a `CORBA::Any` as does the Event Service. In addition, the Notification Service introduces the concept of structured events and sequence events.

Structured Events

Structured events are represented with the `CosNotification::StructuredEvent` type as shown in Figure 3.1. The two main components of a structured event are the *event header* and *event body*. The event header is further sub-divided into a *fixed header* and *variable header*. The fixed header categorizes the event, while the variable header consists of zero or more name-value pairs which specify per-event QoS information. See [1] for complete event header details. The *event body* holds the “interesting” event data in name-value pairs comprising the *filterable fields* and other event data in the opaque *remaining body* field.

Sequence Events

In some instances, it is inefficient to transfer events one-at-a-time. To address this the Notification Service includes support for sequences of structured events on the supplier and consumer side. Suppliers may transfer multiple events to a channel in a single CORBA method invocation; likewise consumers may receive multiple structured events in a single CORBA method call.

Event Translation

The Notification Service does not impose the restriction that peer entities (suppliers and consumers) must deal with the same event type. For example a structured consumer can receive events from an unstructured supplier. Rules exist (see [1]) that define how events are translated into a format suitable for various consumers. Event translation supports configurations like that in Figure 3.2.

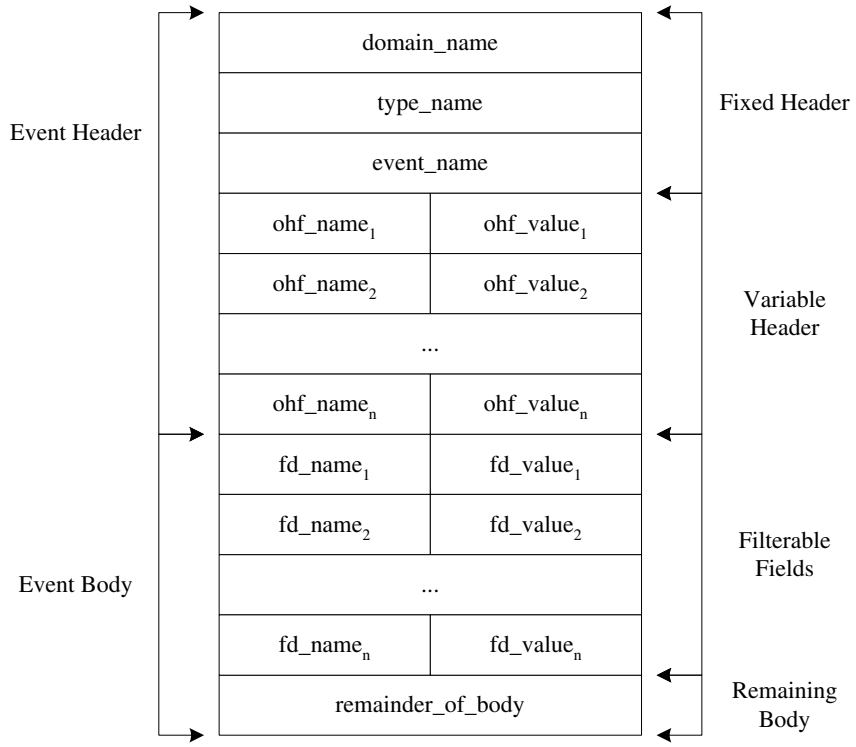

Figure 3.1: CosNotification::StructuredEvent

3.3.5 Filtering

The Notification Service defines a set of interfaces in the `CosNotifyFilter` module which support event filtering. In the same way event channels are created from the `EventChannelFactory`, filters are created from the `FilterFactory`. The default filter factory is available from the `CosNotifyChannelAdmin::EventChannel` interface.

Each filter contains a list of constraints, where each constraint is composed of a list of event types and a single boolean constraint expression (the filter structure is illustrated in Figure 3.3). The constraint expression conforms to some constraint grammar and specifies restrictions based on the data in the event filterable fields. Notify supports the default constraint grammar as specified in [1]. For an event to match a constraint it must match one or more of the event types within that constraint and the constraint expression must evaluate

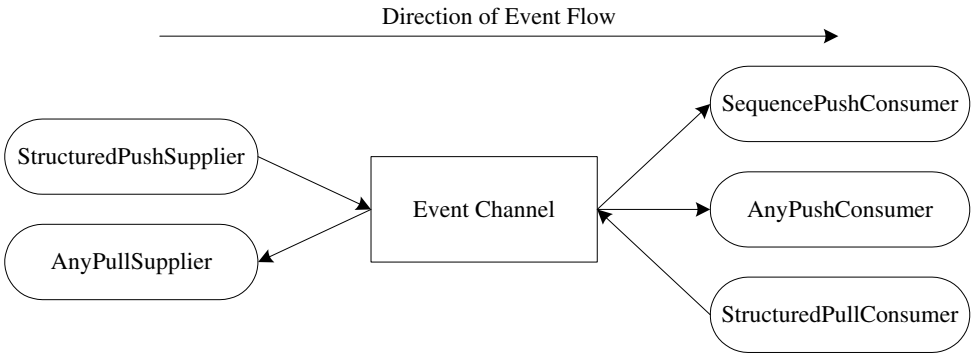

Figure 3.2: Event Translation Example

to true. If a filter contains multiple constraints, OR semantics are applied between the constraints. That is, the boolean result of applying a filter can be expressed as:

$$R_{Filter} = C_1 + C_2 + \ldots + C_N$$

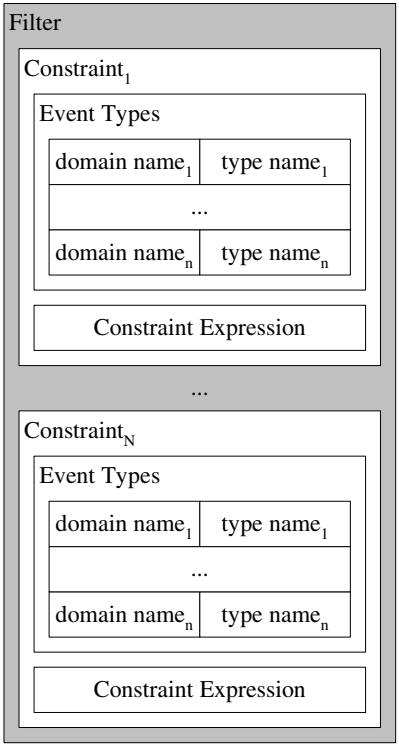

Figure 3.3: Filter Composition

where

R_{Filter}	is the boolean result of applying a filter
C_n, $n=1..N$	is the boolean result of applying constraint n within the filter

A given proxy or admin may have multiple filters associated with it. Again, OR semantics are applied between filter results. That is, the boolean result of applying multiple filters is:

$$R_{AllFilters} = R_{Filter1} + R_{Filter2} + \ldots + R_{FilterN}$$

where:

$R_{AllFilters}$ is the boolean result of applying all filters for an admin or proxy

$R_{Filtern}$, $n=1..N$ is the boolean result of applying filter n

Perhaps the most complicated scenario is when a proxy and its parent admin both have multiple filters associated with them. The filters associated with the admin are applied as described above (using OR semantics). Likewise the filters associated with the proxy are applied (again using OR semantics). Next the results of these two operations are combined. The semantics, AND or OR, of this final operation are specified at the time the admin object was created and is known as the *interfilter group operator*. So, for the configuration depicted in Figure 3.4 the following expression applies

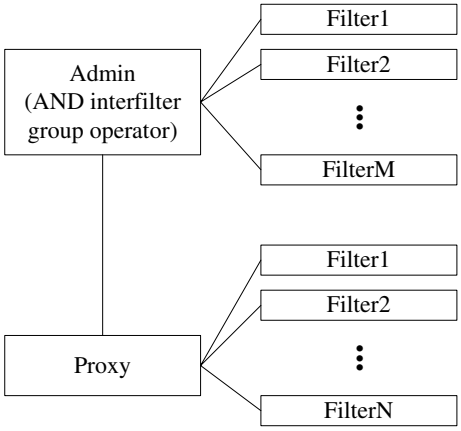

Figure 3.4: Admin and Proxy Filtering

$$R_{Final} = (R_{Filter1} + R_{Filter2} + \ldots + R_{FilterM}) \cdot (R_{Filter1} + R_{Filter2} + \ldots + R_{FilterN})$$

where:

R_{Final} is the boolean result of applying all filters for the admin and proxy

$R_{Filterm}$, $m=1..M$ is the boolean result of applying admin filter m

$R_{Filterm}$, $n=1..N$ is the boolean result of applying proxy filter n

If the OR interfilter group operator is specified during creation of the admin object then the resulting expression is:

$$R_{Final} = (R_{Filter1} + R_{Filter2} + \ldots + R_{FilterM}) + (R_{Filter1} + R_{Filter2} + \ldots + R_{FilterN})$$

Filters can be applied at the supplier and consumer ends of a channel, and at the admin and proxy levels. Also note that a single filter can be associated with multiple admins or proxies. This practice is not recommended, since it can lead to a service which is difficult to manage.

Mapping Filters

Mapping filters allow consumers to affect the priority and lifetime settings of an event. The application of a mapping filter does not actually change any event settings, instead it influences how the consumer perceives the event. For this reason mapping filters can only be added to proxy suppliers and consumer admin objects. The structure of the mapping filter is shown in Figure 3.5.

The IDL interface for mapping filters, `MappingFilter`, is defined in the `CosNotifyFilter` module. Note the similarities between mapping filters and regular filters:

- both have a list of constraints
- within each constraint, there is a constraint expression and list of event types

When a mapping filter is applied to an event each constraint is checked until a match is found or there are no more constraints. If there is a match then the value stored in the *Value* field of the matching constraint is returned to the proxy. This value is used instead of the actual value for the event property. If there is no match then the property value contained in the event is used, unless the event does not specify this property, in which case the mapping filter *Default Value* is used.

3.3.6 Quality of Service

The Notification Service defines standard interfaces for controlling the QoS characteristics of event delivery. QoS is specified on a per-event, per-consumer/supplier, or per-chan-

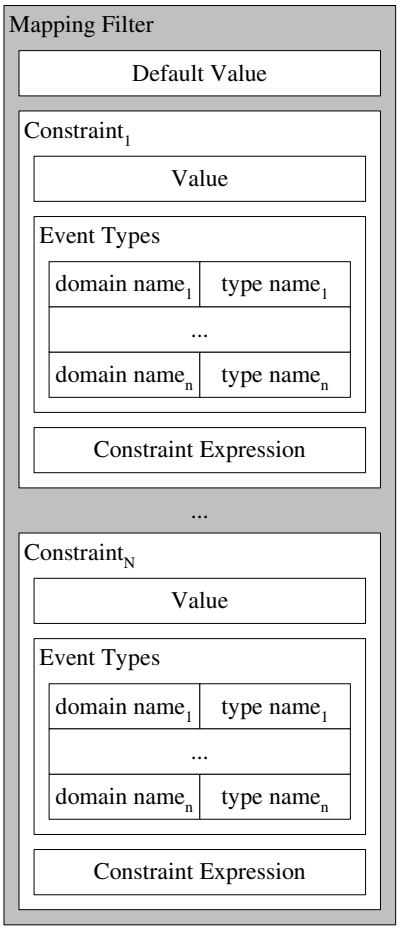

Figure 3.5: Mapping Filter Composition

nel basis. Interfaces which support QoS properties derive from the `CosNotification::QoSAdmin` IDL interface.

Persistence

Perhaps one of the most important QoS properties added by the Notification Service is persistence as it applies to event and connection reliability QoS parameters.

Connection Persistence

Persistent connection reliability refers to ORBACUS NOTIFY's ability to restore all object connections after a service restart. That is, when ORBACUS NOTIFY starts it restores all channels, admins, proxies, and filters to their state at shutdown. In addition ORBACUS NOTIFY also attempts to re-establish communication with any clients that were connected at shutdown.

ORBACUS NOTIFY can also restore connections to restarted clients. These clients must supply a persistent, non-nil object reference when connecting to the proxy.

Note: ORBACUS NOTIFY *does not permit admin and proxy objects to set a connection reliability different than that set on the parent event channel.*

Event Persistence

With connection persistence enabled, ORBACUS NOTIFY also supports event persistence. That is all consumers connected at the time an event is delivered to the channel are guaranteed to receive that event within event expiry limits.

The following briefly describes the available properties.

Event QoS Properties

The following QoS properties are set on a per event basis.

`EventReliability`

Value: `BestEffort,Persistent`

`EventReliability`, when set on a per event basis, sets a different reliability for the target event than that specified at the channel/admin/proxy level. Note that it is not permitted to specify per event `Persistent` event reliability over a channel with `BestEffort` event reliability. By default, the reliability of event delivery is determined by the `EventReliability` setting of the channel.

`Priority`

Value: -32767 <= *priority* <= 32767

The order in which events are delivered to a consumer can be specified based on the priority of an event. The lowest priority is -32,767 and 32,767 is the highest. The default priority is 0.

`Timeout`

Value: `TimeBase::TimeT`

`Timeout` states a relative expiry time after which an event can be discarded. By default, events have no relative expiry time.

`StopTime`

Value: `TimeBase::UtcT`

`StopTime` states an absolute expiry time after which an event can be discarded. By default, events have no absolute expiry time.

`StartTime`

Value: `TimeBase::UtcT`

`StartTime` states an absolute earliest delivery time after which the event can be delivered. The `StartTime` property provides the ability to hold an event until a specified time, and be eligible for delivery only after that time. By default, events are eligible for transmission as soon as they are received by the service.

QoS Properties

The following QoS properties are set on a per channel/admin/proxy basis.

`EventReliability`

Value: `BestEffort,Persistent`

`EventReliability` is set on the channel object and determines whether the delivery of all events on the channel will be `Persistent` or `BestEffort`. The default is `BestEffort.`

`ConnectionReliability`

Value: `BestEffort,Persistent`

`ConnectionReliability` applies to channel, admin, and proxy objects, and the re-establishment of supplier and consumer connections. The default is `BestEffort`.

`MaxEventsPerConsumer`

Value: *events* >= 0

The `MaxEventsPerConsumer` property is used to limit the number of events that will be queued in a ProxySupplier. The default is 0, meaning no limit.

`OrderPolicy`

Value: `AnyOrder`, `FifoOrder`, `PriorityOrder`, `DeadlineOrder`

`OrderPolicy` determines the order in which events are queued for delivery to a consumer. `AnyOrder` means that any ordering policy (`FifoOrder`, `PriorityOrder`, or `DeadlineOrder`) may be used. The default is `PriorityOrder`.

`DiscardPolicy`

Value: `AnyOrder`, `FifoOrder`, `LifoOrder`, `PriorityOrder`, `DeadlineOrder`, `RejectNewEvents`

`DiscardPolicy` applies when a queue reaches a limit specified by `MaxEventsPerConsumer` admin property, and specifies the order in which events should be discarded. The default is `AnyOrder` meaning that any event may be discarded on overflow.

`MaximumBatchSize`

Value: *size* > 0

Indicates the maximum number of events that will be delivered in a sequence of structured events. The default is 1.

`PacingInterval`

Value: `TimeBase::TimeT`

`PacingInterval` is the maximum period of time a channel will collect events into a sequence before delivering the sequence. The default is 0, meaning that a sequence of events is transmitted when ready.

Note: For a more extensive description of the above listed properties please refer to [1].

3.3.7 Proprietary QoS Properties

While the Notification Service specification [1] defines a wide range of QoS properties, there are some important features which remain undefined. For example, although the specification provides QoS properties to control priority, expiry times, and earliest delivery time for events, it does not specify how an event communication failure is handled. Similarly, for pull events, the specification does not define how often the pull should occur. To address these deficiencies, ORBACUS NOTIFY implements a number of proprietary features. The IDL names for these features are specified in the `OBNotify` module.

Properties for Retry Handling of a Failed Event Communication

ORBACUS NOTIFY includes several QoS properties which configure proprietary retry handling facilities. A retry occurs when ORBACUS NOTIFY attempts to "push" an event and receives an exception, thereby prompting it to retry sending the event at specified intervals.

`RetryTimeout`

Value: `TimeBase::TimeT`

The `RetryTimeout` specifies the initial amount of time that ORBACUS NOTIFY waits before trying to resend an event after a communication failure with a client. The default value is 1 second.

`RetryMultiplier`

Value: 1.0 <= *multiplier* <= 2.0

The `RetryMultiplier` is the value by which the current value of the `RetryTimeout` is multiplied to determine the next `RetryTimeout` value. The `RetryMultiplier` may also be used to provide a backoff value if necessary. The default value is 1.0.

`MaxRetryTimeout`

Value: `TimeBase::TimeT`

The `MaxRetryTimeout` property is the maximum value or ceiling that the `RetryTimeout` can have. This property applies to `RetryTimeout` values that are directly assigned by a developer as well as those that are generated from the multiplication of the `RetryMultiplier` and `RetryTimeout`. The default value is 60 seconds.

The relationship among the above properties is defined as follows:

$$RetryTimeout \times RetryMultiplier \leq MaxRetryTimeout$$

`MaxRetries`

Value: *retries* >= 0

The `MaxRetries` value is the maximum number of times that a failed event communication should be retried. Once this number has been reached, the proxy is destroyed and the communication terminated. The default value is 0, meaning unlimited retries.

`RequestTimeout`

Value: `TimeBase::TimeT`

The amount of time permitted for a blocking request on a client to return before a timeout. The default value is 5 seconds.

Other Proprietary QoS Properties

This section describes other proprietary QoS properties available for ORBACUS NOTIFY

`PullInterval`

Value: *interval* >= 0

ORBACUS NOTIFY includes a `PullInterval` property to specify how often events should be pulled from suppliers. This property is applicable to the pull model and enables users to configure the frequency of pull requests made on suppliers. The default value is 1 second.

`RequestTimeout`

Value: `TimeBase::TimeT`

The `RequestTimeout` property specifies the maximum time limit for requests made on pull suppliers and push consumers by their associated proxies. The maximum value for this property is 10 minutes. The default value is 5 seconds.

3.3.8 Administrative Properties

In addition to configurable QoS properties, event channels also support the configuration of certain administrative properties. The following administrative properties, each of type long, are supported by an event channel.

`MaxConsumers`

Value: *consumers* >= 0

The maximum number of consumers that can be connected to a channel at any given time.

`MaxSuppliers`

Value: *suppliers* >= 0

The maximum number of suppliers that can be connected to a channel at any given time.

The default value is 0 for all properties, meaning that no limit applies to that property.

3.3.9 Subscription Sharing

Subscription sharing is a standard mechanism for suppliers to publish the types of events that they will supply and for consumers to subscribe to event types that they wish to receive. The information can be used by suppliers and consumers to decide whether they wish to supply events or consume events on a notification channel.

The Notification Service supports subscription sharing between channels and channel clients through the following interfaces:

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

    }; // NotifyPublish

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

Supplier admins and proxy consumers inherit the `NotifyPublish` interface. Suppliers may use the `offer_change` method to notify the channel that it is about to start supplying new event types or is about to stop supplying an existing type. The channel maintains an aggregate list of all event types currently offered; and when this changes it notifies consumers through the `offer_change` method.

Consumer admins and proxy suppliers inherit the `NotifySubscribe` interface. Consumers may use the `subscription_change` method to subscribe/unsubscribe to a set of channel events. Again, the channel maintains an aggregate list of all subscriptions, and when this changes it notifies suppliers through the `subscription_change` method.

Subscription sharing allows sophisticated suppliers and consumers to dynamically control the types of events that flow through the channel. This can increase channel efficiency since unwanted events are no longer produced.

CHAPTER 4

Programming Example

4.1 Introduction

This chapter describes a set of steps which implement a simple ORBACUS NOTIFY supplier and consumer. The supplier uses the push model to present structured event data to the event channel. Similarly the consumer uses the push model to receive events from the same channel. Each event represents a letter of the alphabet in both upper and lower case forms (see Figure 4.1). Note that this example is taken from the C++ demos which

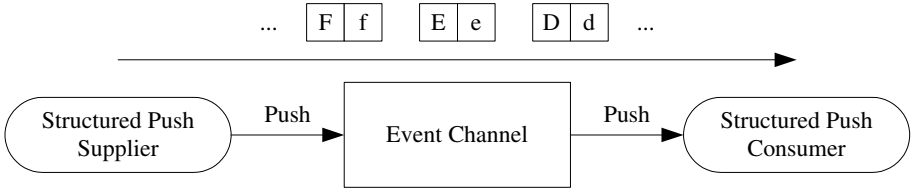

Figure 4.1: ORBACUS NOTIFY **Example**

accompany the ORBACUS NOTIFY distribution (see `notify/demo/simple/StructuredPushSupplier.cpp` and `notify/demo/simple/StructuredPushConsumer.cpp`) or the equivalent Java demos which accompany the ORBACUS NOTIFY Console distribution (`notify/demo/simple/`

`StructuredPushSupplier.java` and `notify/demo/simple/StructuredPushConsumer.java`).

In this example, the supplier and consumer create the channel, admin and proxy objects. Alternatively an ORBACUS NOTIFY client could use an already existing object, either through a published IOR or via the unique ID assigned to such objects within ORBACUS NOTIFY.

For clarity, appropriate exception handling and error checking is not included in the code snippets.

4.2 Connecting to a Notification Channel

This section describes how suppliers and consumers connect to a notification channel so that they may transfer events. Figure 4.2 illustrates how the supplier and consumer connect to an event channel in this example. Each of these steps are described next.

4.2.1 Resolving the Event Channel Factory

Before an application can obtain an event channel it must first resolve the `"NotificationService"` initial reference. The result is an object of type `CosNotifyChannelAdmin::EventChannelFactory`. The C++ and Java code follows:

```
1 // C++
2 CORBA::Object_var obj =
3    orb -> resolve_initial_references("NotificationService");
4
5 CosNotifyChannelAdmin::EventChannelFactory_var eventChannelFactory
6    = CosNotifyChannelAdmin::EventChannelFactory::_narrow(obj);
```

```
1 // Java
2 org.omg.CORBA.Object obj =
3    orb.resolve_initial_references("NotificationService");
4
5 EventChannelFactory eventChannelFactory =
6    EventChannelFactoryHelper.narrow(obj);
```

2-3 Resolve the “NotificationService” initial reference.

5-6 Narrow the reference to the appropriate type.

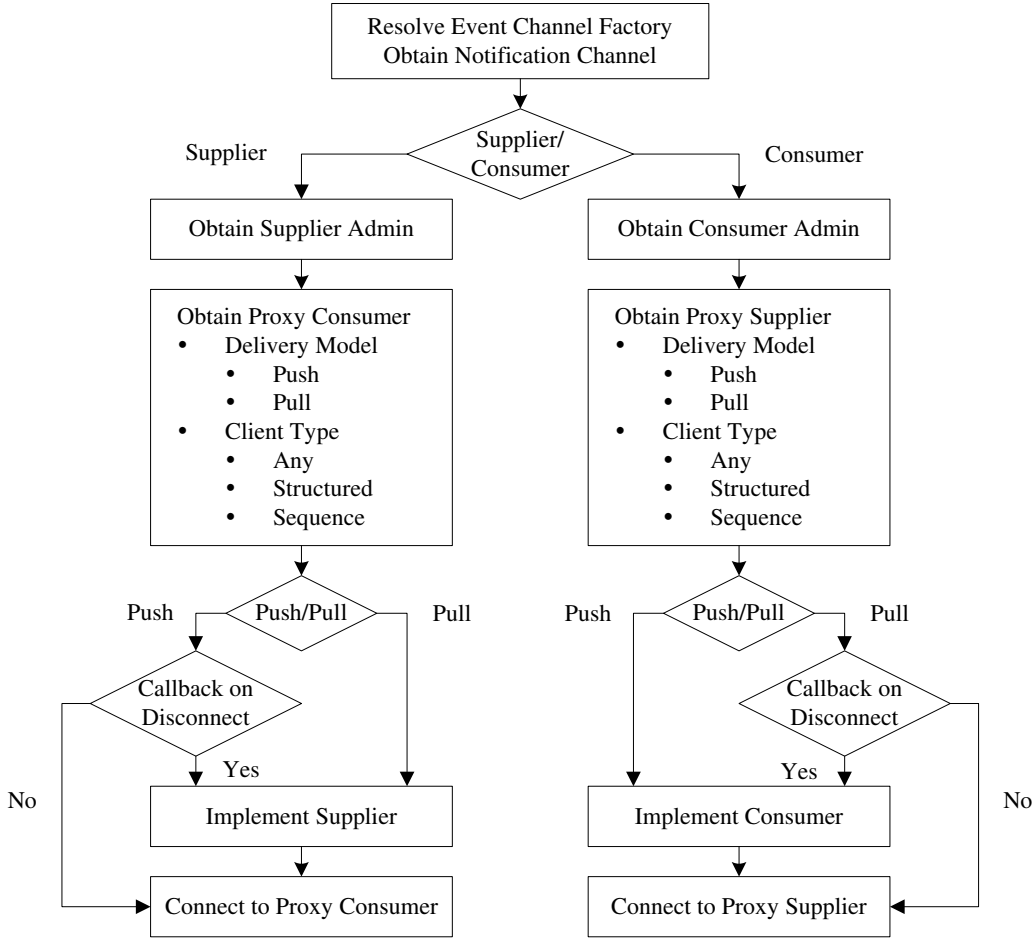

Figure 4.2: Connecting to a Notification Channel

4.2.2 Obtaining an Event Channel

The object reference to the `CosNotifyChannelAdmin::EventChannelFactory` is used to create an event channel. Another option is to ask for an existing channel using an ID previously assigned by ORBACUS NOTIFY:

```
// IDL
```

```
interface EventChannelFactory
{
    ...
    EventChannel get_event_channel(in ChannelID id)
        raises(ChannelNotFound);
    ...
};
```

This example creates the channel, if necessary, and publishes the IOR of the newly created channel[1], otherwise an already published IOR is used to get a channel reference. Note that only one of the supplier or consumer actually creates the channel, depending on which is started first. It then publishes the IOR for the newly created channel for use by its peer.

In C++ the channel is created as follows:

```
1 // C++
2 CosNotification::QoSProperties initialQoS;
3 CosNotification::AdminProperties initialAdmin;
4 CosNotifyChannelAdmin::ChannelID channelId;
5 CosNotifyChannelAdmin::EventChannel_var eventChannel =
6     eventChannelFactory -> create_channel(initialQoS,
7                                           initialAdmin,
8                                           channelId);
```

In Java:

```
1 // Java
2 Property[] initialQoS = new Property[0];
3 Property[] initialAdmin = new Property[0];
4 org.omg.CORBA.IntHolder channelId = new org.omg.CORBA.IntHolder();
5 EventChannel eventChannel =
6     eventChannelFactory.create_channel(initialQoS,
7                                        initialAdmin,
8                                        channelId);
```

2-3 Create empty property sequences for QoS and Channel Administration. To specify properties other than the default, add the appropriate name-value pairs to these sequences. For this example the default properties are sufficient.

4 The unique channel ID assigned by ORBACUS NOTIFY is passed back in the `channelId` parameter.

1. For this simple example the IOR is published in a file. See the C++ or Java demos for details.

5-8 Use the event channel factory to create a new channel.

Alternatively, a channel may be obtained from an IOR. In C++:

```
// C++
CORBA::Object_var obj = ... // Get reference to the channel
CosNotifyChannelAdmin::EventChannel_var eventChannel =
    CosNotifyChannelAdmin::EventChannel::_narrow(obj);
```

And in Java:

```
// Java
org.omg.CORBA.Object obj = ... // Get reference to the channel
EventChannel eventChannel = EventChannelHelper.narrow(obj)
```

The code presented so far applies equally to supplier and consumer applications using either the push or pull model. Connecting the supplier and consumer is discussed next.

4.2.3 Connecting a Supplier

This section describes how to connect an event supplier to an event channel. Figure 4.3 illustrates the steps.

Supplier Admin

The first step in connecting a supplier is to obtain a supplier admin object. All event channels come with two read only attributes: `default_supplier_admin` and `default_consumer_admin`.

```
// IDL
interface EventChannel
{
    ...
    readonly attribute ConsumerAdmin default_consumer_admin;
    readonly attribute SupplierAdmin default_supplier_admin;

    ...
};
```

This example uses the default admin objects:

```
// C++
CosNotifyChannelAdmin::SupplierAdmin_var supplierAdmin =
    eventChannel -> default_supplier_admin();

// Java
```

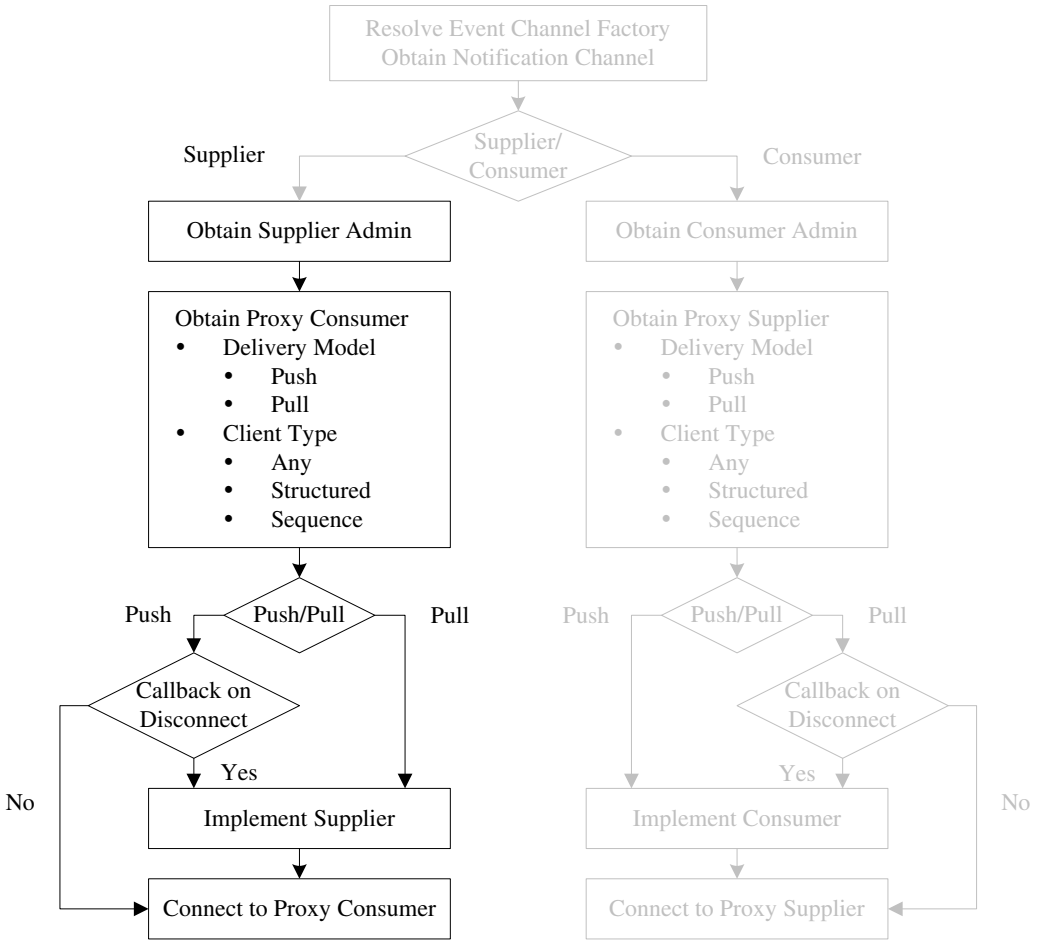

Figure 4.3: Connecting a Supplier to a Notification Channel

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

Supplier applications may also create a new supplier admin using the following:

```
// IDL
EventChannel
{
```

```
    ...
    SupplierAdmin new_for_suppliers(
        in InterFilterGroupOperator op,
        out AdminID id );
    ...
};
```

or use an admin with a given `AdminID`. Note that `AdminID` is a unique ID assigned by ORBACUS NOTIFY.

```
// IDL
EventChannel
{
    ...
    SupplierAdmin get_supplieradmin ( in AdminID id )
        raises (AdminNotFound);
    ...
};
```

Proxy Consumer

The next step in connecting to an event channel is to obtain the proper proxy consumer from the supplier admin. This is the point at which the application specifies the delivery model and type of events it will supply. This example uses the push delivery model and structured events. The C++ code looks like:

```
//C++
CosNotifyChannelAdmin::ProxyID proxyId;

CosNotifyChannelAdmin::ProxyConsumer_var proxyConsumer =
    supplierAdmin -> obtain_notification_push_consumer(
        CosNotifyChannelAdmin::STRUCTURED_EVENT, proxyId);

CosNotifyChannelAdmin::StructuredProxyPushConsumer_var
    structuredProxyPushConsumer =
       CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(
            proxyConsumer);
```

And in Java:

```
// Java
org.omg.CORBA.IntHolder proxyId = new org.omg.CORBA.IntHolder();

ProxyConsumer proxyConsumer =
    supplierAdmin.obtain_notification_push_consumer(
```

```
6          ClientType.STRUCTURED_EVENT, proxyId);
7
8  StructuredProxyPushConsumer
9      structuredProxyPushConsumer =
10         StructuredProxyPushConsumerHelper.narrow(
11             proxyConsumer);
```

2 Variable to hold the ID later assigned to the proxy by ORBACUS NOTIFY.

4-6 Obtain a push consumer, specifying the type. This example wants a structured event push consumer. Valid types are `ANY_EVENT`, `STRUCTURED_EVENT`, `SEQUENCE_EVENT`.

8-11 Narrow the proxy consumer to the appropriate type specified in the previous call.

Equivalent objects and methods exist for pull model suppliers.

Connecting to a Proxy

The final step in connecting a supplier to an event channel is to connect to the proxy. Each of the various proxy types implement their own connect method. A proxy of type `CosNotifyChannelAdmin::StructuredProxyPushConsumer` is used in this example:

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

A supplier registers itself with a proxy when it invokes the appropriate connect method. If the supplier wants notification of either of the following:

- when it is about to be disconnected
- when there is a change in the set of events to which consumers are currently subcribed

it must implement the appropriate CORBA servant and pass it as an argument in the connect call. In this case the supplier must also assume the role of CORBA server.

The example supplier is not interested in these notifications so it passes a nil argument during the connect call:

```
// C++
```

```
structuredProxyPushConsumer -> connect_structured_push_supplier(
                CosNotifyComm::StructuredPushSupplier::_nil());


// Java
structuredProxyPushConsumer.
                    connect_structured_push_supplier(null);
```

4.2.4 Connecting a Consumer

This section describes how to connect to an event channel so that an application may receive events. Figure 4.4 outlines the process of connecting a consumer to an event channel.

Consumer Admin

The first step in connecting a consumer is to obtain a consumer admin. As mentioned earlier each event channel comes with default supplier and admin objects. The example consumer uses the default consumer admin:

```
// C++
CosNotifyChannelAdmin::ConsumerAdmin_var consumerAdmin =
   eventChannel -> default_consumer_admin();

// Java
ConsumerAdmin consumerAdmin =
   eventChannel.default_consumer_admin();
```

As with supplier applications, consumers may also create a new consumer admin object using the following:

```
// IDL
EventChannel
{
   ...
   ConsumerAdmin new_for_consumers(
      in InterFilterGroupOperator op,
      out AdminID id );
   ...
};
```

or use an admin with a given ID (of type `CosNotifyChannelAdmin::AdminID`). Note that this is a unique ID assigned by Notify.

```
// IDL
```

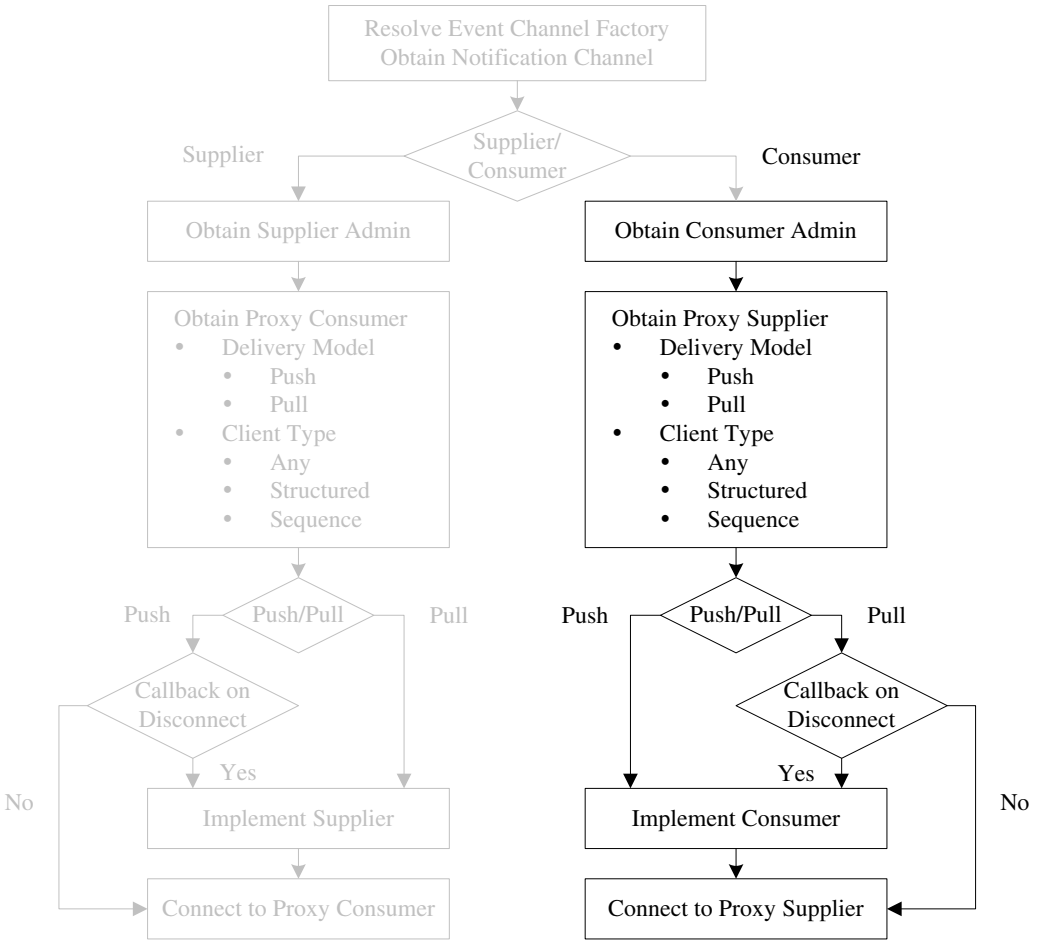

Figure 4.4: Connecting a Consumer to a Notification Channel

```
EventChannel
{
    ...
    ConsumerAdmin get_consumeradmin ( in AdminID id )
        raises (AdminNotFound);
    ...
};
```

Proxy Supplier

The next step in connecting a consumer to an event channel is to obtain the appropriate proxy supplier from the consumer admin object. Like the supplier example, this is where the consumer specifies the delivery model and type of events it wishes to receive.

It is important to note that the type of proxies used by suppliers and consumers are independent of each other. Hybrid delivery models are supported, for example a pull consumer can receive events from a push supplier. Also the type of event specified by the proxies are independent due to the event translation capabilities of the channel. For example, structured events inserted into a `CORBA::Any` by the supplier are received as structured events by a structured consumer.[1]

This example, like the supplier, uses the push delivery model and structured events. The corresponding C++ code is:

```
// C++
CosNotifyChannelAdmin::ProxyID proxyId;

CosNotifyChannelAdmin::ProxySupplier_var proxySupplier =
   consumerAdmin -> obtain_notification_push_supplier(
      CosNotifyChannelAdmin::STRUCTURED_EVENT, proxyId);

CosNotifyChannelAdmin::StructuredProxyPushSupplier_var
   structuredProxyPushSupplier =
      CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(
         proxySupplier);
```

And in Java:

```
// Java
org.omg.CORBA.IntHolder proxyId = new org.omg.CORBA.IntHolder();

ProxySupplier proxySupplier =
   consumerAdmin.obtain_notification_push_supplier(
      ClientType.STRUCTURED_EVENT, proxyId);

StructuredProxyPushSupplier
   structuredProxyPushSupplier =
```

1. Try running different combinations of the demo suppliers and consumers which accompany the ORBACUS NOTIFY distribution (see `notify/demo/simple`). For example try running the `SequencePullSupplier` and the `AnyPushConsumer`.

```
10        StructuredProxyPushSupplierHelper.narrow(
11           proxySupplier);
```

2 Variable to hold the ID later assigned to the proxy by ORBACUS NOTIFY.

4-6 Obtain a proxy push supplier specifying the type. This example wants a structured event proxy push supplier. Valid types are ANY_EVENT, STRUCTURED_EVENT, SEQUENCE_EVENT.

8-11 Narrow the proxy supplier to the appropriate type specified in the previous call.

Equivalent objects and methods exist for pull model consumers.

Connecting to a Proxy

The final step in connecting a consumer to an event channel is to connect to the proxy. This is similar to connecting the supplier with one major difference: a push consumer must implement the appropriate CORBA servant to support the event delivery. A push consumer must assume the role of CORBA server since it has to process incoming requests, namely handle events pushed by the channel. The implementation of the push consumer servant is discussed next.

Implementing the Servant

```
1  // C++
2  class StructuredPushConsumer_impl :
3     public POA_CosNotifyComm::StructuredPushConsumer,
4     public PortableServer::RefCountServantBase
5  {
6     CORBA::ORB_var orb_;
7     PortableServer::POA_var poa_;
8
9  public:
10    StructuredPushConsumer_impl(
11       CORBA::ORB_ptr orb, PortableServer::POA_ptr poa) :
12       orb_(CORBA::ORB::_duplicate(orb)),
13       poa_(PortableServer::POA::_duplicate(poa))
14    {
15    }
16
17    virtual ~StructuredPushConsumer_impl()
18    {
19    }
20
21    void
```

```
22     push_structured_event(
23        const CosNotification::StructuredEvent& event)
24     {
25        cout << "Pushed..." << endl;
26        if(DisplayEvent(event))
27           throw CosEventComm::Disconnected();
28     }
29
30     void
31     disconnect_structured_push_consumer()
32     {
33        PortableServer::ObjectId_var oid =
34           poa_ -> servant_to_id(this);
35        poa_ -> deactivate_object(oid.in());
36
37        orb_ -> shutdown(false);
38     }
39
40     void
41     offer_change(const CosNotification::EventTypeSeq& added,
42                  const CosNotification::EventTypeSeq& removed)
43     {
44        // Event offering has changed
45     }
46  };
```

2-4 New class defining our reference counted servant. Note the derivation from `POA_CosNotifyComm::StructuredPushConsumer` which is generated by the IDL compiler from CosNotifyComm.idl.

6-7 Keep a reference to the ORB and the POA.

10-19 Constructor and destructor. Store our reference to the ORB and POA in `_var` types for automatic memory management.

21-28 Implement the `push_structured_event` method. This method is invoked each time the channel pushes an event; in this example the consumer displays the event. The `DisplayEvent`[1] routine returns true when an event containing the last letter of the alphabet is received, prompting the consumer to disconnect from the channel.

30-38 On disconnection by the channel, disconnect the servant and end the process. Invoking `shutdown()` causes the ORB's `run()` method to return.

1. For the details of DisplayEvent() see any of the demos which accompany the ORBACUS NOTIFY distribution in `notify/demo/simple`).

40-45 Not implemented in this example. This method communicates changes in the event type offering on the channel. Sequences of event types being added and event types being removed are passed as parameters.

The corresponding Java code is presented below:

```
// Java
class StructuredPushConsumer_impl extends
    StructuredPushConsumerPOA
{
    private ORB orb_;
    private POA poa_;

    StructuredPushConsumer_impl(ORB orb, POA poa)
    {
        orb_ = orb;
        poa_ = poa;
    }

    public void
    push_structured_event(StructuredEvent event)
        throws org.omg.CosEventComm.Disconnected
    {
        System.out.println("Pushed...");
        if(StructuredPushConsumer.displayEvent(event))
            throw new org.omg.CosEventComm.Disconnected();
    }

    public void
    disconnect_structured_push_consumer()
    {
        byte[] oid =poa_.servant_to_id(this);
        poa_.deactivate_object(oid);

        orb_.shutdown(false);
    }

    public void
    offer_change(EventType[] added, EventType[] removed)
    {
        // Event offering has changed
    }
}
```

2-3 New class defining our servant. Note the derivation from `StructuredPushConsumerPOA` which is generated by the IDL compiler from CosNotifyComm.idl.

5-6 See *6-7* above.

8-12 Constructor.

14-21 See *21-28* above.

23-30 See *30-38* above.

32-36 See *40-45* above.

Once the servant is implemented it is registered with the proxy supplier:

```
// C++
CosNotifyComm::StructuredPushConsumer_var structuredPushConsumer =
   new StructuredPushConsumer_impl(orb, poa);

structuredProxyPushSupplier ->
   connect_structured_push_consumer(structuredPushConsumer);


// Java
StructuredPushConsumer_impl structuredPushConsumer =
   new StructuredPushConsumer_impl(orb, poa);

structuredProxyPushSupplier.connect_structured_push_consumer(
   structuredPushConsumer);
```

All that remains is to activate the POA and run the ORB, and the consumer is ready to receive events.

4.3 Supplying Events

The mechanism of supplying events to a notification channel depends on the delivery model. The ORBACUS NOTIFY C++ and Java demos implement push and pull suppliers with any, structured, and sequence events.

4.3.1 Push Supplier

Implementing a push supplier is relatively easy since no CORBA servants are required for the most basic applications[1]. Once connected to the proxy, the application can immedi-

ately start supplying events. This example pushes events within the `main` subroutine as shown below:

```
// C++
const int numChars = 26;
for(int i = 0 ; i < numChars ; ++i)
{
   cout << "Pushing..." << endl;
   CosNotification::StructuredEvent_var event =
      CreateNewEvent(i);

   structuredProxyPushConsumer -> push_structured_event(*event);
}
```

And in Java:

```
// Java
final int numChars = 26;
for(int i = 0 ; i < numChars ; ++i)
{
   System.out.println("Pushing...");
   StructuredEvent event = createNewEvent(orb, i);

   structuredProxyPushConsumer.push_structured_event(event);
}
```

The different types of push suppliers have similar but distinct IDL interfaces. The IDL for the structured push supplier is:

```
// IDL
interface StructuredPushSupplier : NotifySubscribe
{
   void disconnect_structured_push_supplier();
};
```

Our example does not implement this interface for reasons stated earlier.

4.3.2 Pull Supplier

Unlike the push supplier, the pull supplier assumes a passive role in event delivery. The push supplier is active in that it initiates event delivery on the channel. Conversely, the pull supplier is passive and has events pulled from it by the channel. For this reason the pull

1. A servant is required if the supplier is interested in knowing when it is disconnected or when the channel subscription information changes.

supplier must implement a servant which incarnates a CORBA object capable of accepting requests from ORBACUS NOTIFY. Separate, but similar, IDL interfaces exist for the any, structured and sequence pull suppliers. The IDL for the structured pull supplier is given below.

```
// IDL
interface StructuredPullSupplier : NotifySubscribe
{
   CosNotification::StructuredEvent pull_structured_event()
      raises(CosEventComm::Disconnected);

   CosNotification::StructuredEvent try_pull_structured_event(
      out boolean has_event)
      raises(CosEventComm::Disconnected);

   void disconnect_structured_pull_supplier();
};
```

The blocking `pull_structured_event()` and non-blocking `try_pull_structured_event()` are the methods which retrieve events from the supplier.

4.4 Consuming Events

Like supplying events, receiving events varies with the selected delivery model. The ORBACUS NOTIFY C++ and Java demos implement push and pull consumers for any, structured and sequence events.

4.4.1 Push Consumer

The push consumer is passive and has events pushed on it by ORBACUS NOTIFY. As such it needs to implement the appropriate servant. As with the suppliers, there are separate IDL interfaces for the different push consumers (any, structured, sequence). Below is the IDL for the structured push consumer.

```
// IDL
interface StructuredPushConsumer : NotifyPublish
{
   void push_structured_event(
      in CosNotification::StructuredEvent notification)
      raises(CosEventComm::Disconnected);

   void disconnect_structured_push_consumer();
};
```

It is in the servant's implementation of `push_structured_event()` that events are received by the push consumer.

4.4.2 Pull Consumer

Compared to the push consumer, the pull consumer is the easier to implement and may be likened to the push supplier. The most basic pull consumer need not implement a servant but may directly invoke the methods of the proxy pull supplier interface. The any, structured, and sequence pull suppliers have separate IDL interfaces. The structured pull consumer IDL is given below:

```
// IDL
interface StructuredPullConsumer : NotifyPublish
{
   void disconnect_structured_pull_consumer();
};
```

4.5 Filtering

So far this chapter has covered the details of connecting to an event channel and event delivery mechanisms. One of the powerful features of ORBACUS NOTIFY is the ability to filter events on both the supplier and consumer side. In particular, filters may be applied to supplier and consumer admins and to supplier and consumer proxies. This section extends the structured push consumer example by applying a filter to the supplier proxy (`FilteredConsumer.cpp` and `FilteredConsumer.java` in the C++ and Java demos implement event filtering).

The steps in applying a filter are illustrated in Figure 4.5.

In this example, the filter object is treated much like an event channel in that it is not necessarily created during each execution of the demo. If the demo application determines that it must create a filter, it does so and publishes the IOR for the filter. Subsequent executions of the demo then attempt to re-use this filter. Obtaining a filter from its IOR is straightforward:

```
// C++
CORBA::Object_var obj = ... // Get object from filter IOR
CosNotifyFilter::Filter_var filter =
   CosNotifyFilter::Filter::_narrow(obj);


// Java
org.omg.CORBA.Object obj = ... // Get object from Filter IOR
```

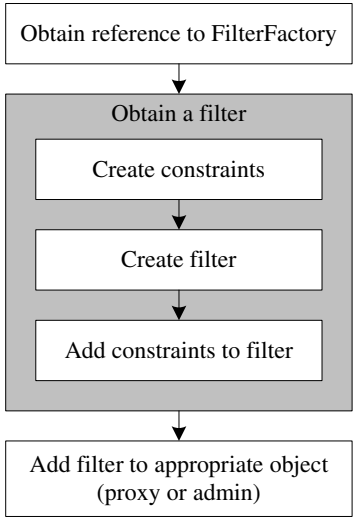

Figure 4.5: Applying a Filter

```
Filter filter = FilterHelper.narrow(obj);
```

4.5.1 Demo Event Structure

The structure of the demo events (see Figure 4.6) is presented before discussing filter creation. .

For demonstration purposes all events share the same `domain_name` and `type_name` field values. The `event_name` field is a concatenation of the filterable field values. The *Variable Header* and *Remaining Body* of the event structure are left empty. The *Filterable Fields* contain two name-value pairs for the lower and upper case versions of alphabetic character.

4.5.2 Obtaining a Reference to the Filter Factory

The first step in applying a filter is to obtain a reference to the Default Filter Factory. Note that every object of type `CosNotifyChannelAdmin::EventChannel` includes a reference to the DefaultFilterFactory:

```
// IDL
interface EventChannel : ...
```

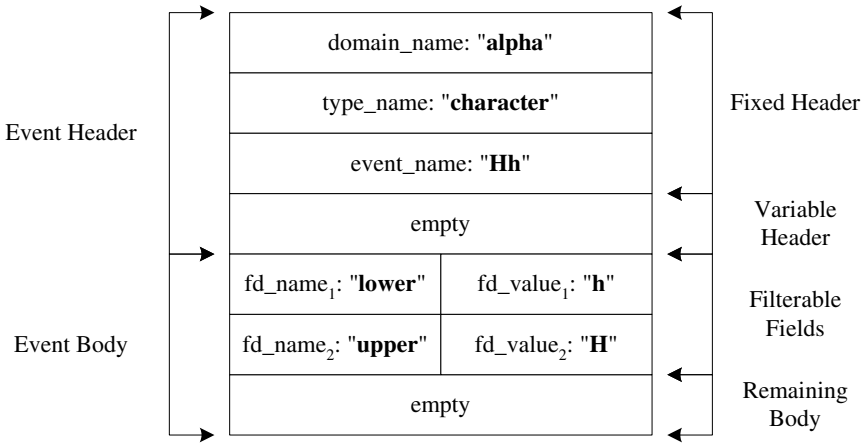

Figure 4.6: Demo Event Structure

```
{
   ...
   readonly attribute CosNotifyFilter::FilterFactory
      default_filter_factory;
...
};
```

In C++ the reference is obtained as follows:

```
// C++
CosNotifyFilter::FilterFactory_var filterFactory =
   eventChannel -> default_filter_factory();
```

And in Java:

```
// Java
FilterFactory filterFactory =
   eventChannel.default_filter_factory();
```

4.5.3 Obtaining a Filter

This section presents the creation of a simple filter, as implemented by the `FilteredConsumer` demo.

Create Filter Constraints

Creating filter constraints involves populating a sequence of type `CosNotifyFilter::ConstraintExpSeq`. The details are presented below.

```
1  // C++
2  const CORBA::ULong numConstraints = 5;
3  const char* constraintStrings[] =
4  {
5     "$upper == 'A'",
6     "$lower == 'e'",
7     "$lower == 'i'",
8     "$upper == 'O'",
9     "$upper == 'U'"
10 };
11
12 CosNotifyFilter::ConstraintExpSeq constraints(numConstraints);
13 constraints.length(numConstraints);
14
15 for(CORBA::ULong i = 0 ; i < numConstraints ; ++i)
16 {
17    constraints[i].event_types.length(1);
18    constraints[i].event_types[0].domain_name =
19       CORBA::string_dup("*");
20
21    constraints[i].event_types[0].type_name =
22       CORBA::string_dup("*");
23
24    constraints[i].constraint_expr =
25       CORBA::string_dup(constraintStrings[i]);
26 }
```

2-10 Constraint expressions. In this example, events which represent vowels are “interesting”. The constraint `"$upper == 'A'"` can be interpreted as: match events which have a filterable field named “upper” and a value of “A”.

12-13 Initialize the sequence to hold `numConstrainsts` expressions.

15 Iterate over the `constraintStrings` array, assigning each element to a separate constraint expression.

17-22 Event types are characterized by the `domain_name` and `type_name` fields. A constraint is the intersection of a single constraint expression and one or more event types. For this example we are only interested in events with filterable data section elements that satisfy our constraint expression. Any event type will satisfy these constraints.

24-25 Specify the constraint expression.

```
1  // Java
2  final int numConstraints = 5;
3  String[] constraintStrings =
4  {
5     "$upper == 'A'",
6     "$lower == 'e'",
7     "$lower == 'i'",
8     "$upper == 'O'",
9     "$upper == 'U'"
10 };
11
12 ConstraintExp[] constraints =
13    new ConstraintExp[numConstraints];
14
15 for(int i = 0 ; i < numConstraints ; ++i)
16 {
17    EventType eventType = new EventType();
18    eventType.domain_name = "*";
19    eventType.type_name = "*";
20
21    EventType[] eventTypes = new EventType[1];
22    eventTypes[0] = eventType;
23
24    ConstraintExp constraint = new ConstraintExp();
25    constraint.event_types = eventTypes;
26    constraint.constraint_expr = constraintStrings[i];
27
28    constraints[i] = constraint;
29 }
```

2-15 See 2-15 above.

17-26 See 17-22 above.

28 See 24-25 above.

Create Filter

Creating a filter is straightforward:

```
// C++
filter = filterFactory -> create_filter("EXTENDED_TCL");
```

```
// Java
filter = filterFactory.create_filter("EXTENDED_TCL");
```

The single argument to the `create_filter()` method specifies the constraint grammar. This example uses `"EXTENDED_TCL"` which is the default grammar supported by all compliant notification services.

Add Constraints to the Filter

Once the filter and constraints are available, the constraints are added to the filter. Again this is straightforward:

```
// C++
CosNotifyFilter::ConstraintInfoSeq_var info =
   filter -> add_constraints(constraints);
```

```
// Java
ConstraintInfo[] info = filter.add_constraints(constraints);
```

The return value of the `add_constraints()` operation is a sequence in which each element contains one of the input constraint expressions and the unique identifier for that expression assigned by ORBACUS NOTIFY.

4.5.4 Adding a Filter to an Admin or Proxy

The IDL interfaces:

```
CosNotifyChannelAdmin::ProxyConsumer
CosNotifyChannelAdmin::ProxySupplier
CosNotifyChannelAdmin::ConsumerAdmin
CosNotifyChannelAdmin::SupplierAdmin
```

all inherit the `CosNotifyFilter::FilterAdmin` interface and can have filter objects associated with them. In this example the filter is added on the consumer side by associating it with the supplier proxy. Adding a filter to the proxy looks like:

```
// C++
CosNotifyChannelAdmin::ProxySupplier_var proxySupplier = ...

...

proxySupplier -> add_filter(filter);
```

```
// Java
ProxySupplier proxySupplier = ...

...

proxySupplier.add_filter(filter);
```

The `add_filter()` operation adds the given filter to the list of filter objects already associated with the target proxy or admin object. It returns an ID, of type `CosNotifyFilter::FilterID`, which is unique amongst all filter objects associated with the particular target proxy or admin. Note that the scope of a filter ID is limited to the scope of the admin or proxy to which the filter is assigned.

4.5.5 Destroying a Filter

The `CosNotifyFilter::Filter` interface includes a method, `destroy()`, which destroys the target filter object. Filters are not strictly owned by a single admin or proxy object. Rather a filter is created from a filter factory and may be added to one or more admin or proxy objects. For this reason, clients must be careful when destroying a filter object, as it may be referenced by other admins and/or proxies within the service. It is recommended that filters not be shared amongst admins or proxies.

This example does not destroy the filter. Rather its IOR is published and used to locate the filter object on subsequent executions of the `FilteredConsumer` example.

4.6 Disconnecting from a Notification Channel

When a supplier or consumer wishes to disconnect from an event channel it simply disconnects its proxy object. The example structured supplier implementation disconnects as follows:

```
// C++
CosNotifyChannelAdmin::StructuredProxyPushConsumer_var
   structuredProxyPushConsumer = ...

...

structuredProxyPushConsumer ->
   disconnect_structured_push_consumer();
```

And in Java:

```
// Java
StructuredProxyPushConsumer structuredProxyPushConsumer = ...
```

```
...

structuredProxyPushConsumer.
   disconnect_structured_push_consumer();
```

LIkewise for the structured push consumer:

```
// C++
CosNotifyChannelAdmin::StructuredProxyPushSupplier_var
   structuredProxyPushSupplier = ...

...

structuredProxyPushSupplier ->
   disconnect_structured_push_supplier();
```

And in Java:

```
// Java
StructuredProxyPushSupplier structuredProxyPushSupplier = ...

...

structuredProxyPushSupplier.
   disconnect_structured_push_supplier();
```

Note that disconnecting a proxy effectively destroys the target proxy object.

Note: The `CosNotifyChannelAdmin::EventChannel`, `CosNotifyChannelAdmin::SupplierAdmin` *and* `CosNotifyChannelAdmin::ConsumerAdmin` *all support the* `destroy()` *operation. Care should be taken when invoking this method since it destroys the target object and all objects it manages. For example, destroying an admin will destroy all proxies managed by that admin, potentially cutting off active communication channels. Similarly, destroying a channel destroys all admins and proxies associated with that channel.*

4.6.1 Disconnecting Passive Clients

Disconnecting from a passive client (push consumer or pull supplier) is not as straight forward as disconnecting from an active client. In the demo examples, the passive servants disconnect by throwing the `CosEventComm::Disconnected` exception from the `push` method when it detects the last event has been received. On receipt of this exception,

ORBACUS NOTIFY invokes the appropriate servant disconnect method which initiates client process termination.

4.7 *Building* ORBACUS NOTIFY *Clients*

The following sections describe how to build ORBACUS NOTIFY clients.

4.7.1 Compiling and Linking C++ Clients

Compiling and linking is to a large degree compiler- and platform-dependent. Many compilers require unique options to generate correct code. ORBACUS NOTIFY clients, at a minimum, must link with the following:

- ORBACUS NOTIFY library - `libCosNotify.a` (Unix) or `CosNotify.lib` (Windows)
- ORBACUS library - `libOB.a` (Unix) or `ob.lib` (Windows)

See the ORBACUS manual and README files which accompany the ORBACUS distribution for various platform-specific compilation instructions.

4.7.2 Compiling Java Clients

Note: The ORBACUS NOTIFY *Java classes are available for download with the* ORBACUS NOTIFY *Console distribution.*

Ensure that the `CLASSPATH` environment variable includes the following:

- ORBACUS NOTIFY Java classes, that is the `OBNotify.jar` file
- ORBACUS for Java classes, that is the `OB.jar` file.

If using the Unix Bourne shell or a compatible shell, this is accomplished with the following commands:

```
CLASSPATH=notify_directory/lib/OBNotify.jar: \
          orbacus_directory/lib/OB.jar:$CLASSPATH
export CLASSPATH
```

Replace `notify_directory` with the name of the directory where ORBACUS NOTIFY is installed; and replace `orbacus_directory` with the name of the directory where ORBACUS is installed.

If running ORBACUS on a Windows-based system, use the following command within the Windows command interpreter:

```
set CLASSPATH=notify_directory\lib\OBNotify.jar; \
             orbacus_directory\lib\OB.jar;%CLASSPATH%
```

Note that for Windows the delimiter is “;” and not “:”.

CHAPTER 5

ORBacus Notify Console

The ORBACUS NOTIFY Console supports the management of all aspects of ORBACUS NOTIFY. The ORBACUS NOTIFY Console includes the following functionality:

- Complete administration of channels, admins and proxies
- QoS configuration at the channel, admin and proxy levels
- Administration of filters
- Administration of mapping filters
- Administration of subscription sharing

5.1 *The Main Window*

The ORBACUS NOTIFY Console main window is shown in 5.1. It contains the following

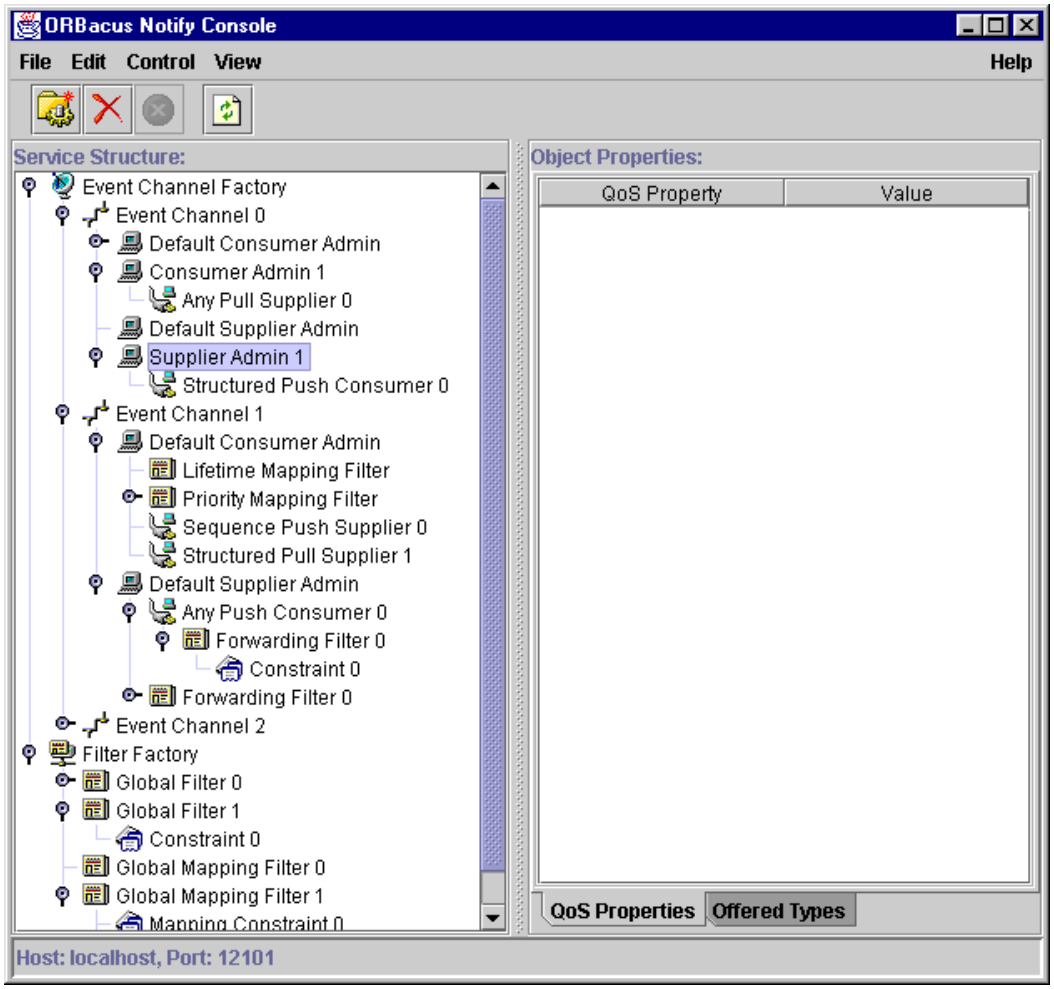

Figure 5.1: The ORBACUS NOTIFY Console Main Window

elements:

Menu bar	Provides access to all the application features.
Toolbar	Shortcuts for the most common menu commands.
Service Structure	Displays the list of configured components in ORBACUS NOTIFY.
Object Properties	Displays the current property settings for the object selected in the Service Structure tree.
Status bar	Displays the host and port at which the console is connected to ORBACUS NOTIFY and also displays information regarding currently executing operations.

5.2 The ORBacus Notify Console Menus

5.2.1 The File Menu

The File Menu contains operations that manage the console windows.

New Window	Creates a new console window connected to the same instance of ORBACUS NOTIFY.
Close	Closes the current console window.
Save IOR...	Writes the IOR for the selected object to a user specified file.
Quit	Quits the application.

5.2.2 The Edit Menu

The Edit Menu contains context sensitive operations which administer the various objects within ORBACUS NOTIFY. These objects include channels, admins, proxies and filters

Create...	Create a new object from the selected factory. In this context the term factory refers to any object which includes factory methods for the creation of other objects. For example an admin object is a factory for both proxy and filter creation.
Destroy	Destroys the selected object.
Properties	Displays a properties dialog for the selected object.

5.2.3 The Control Menu

This menu contains operations which control the operation of ORBACUS NOTIFY.

Shutdown	Shutdown ORBACUS NOTIFY.
Suspend	This operation is available for proxy push supplier and proxy pull consumer objects. It interrupts event flow between the selected proxy and the connected supplier or consumer.
Resume	This operation causes previously suspended proxies to resume pushing or pulling events.

5.2.4 The View Menu

This menu contains operations which allow the user to configure the console display.

Show ToolBar	Toggles between a visible and hidden toolbar.
Show StatusBar	Toggles between a visible and hidden statusbar.
Explicit Refresh	Toggles the refresh mode of the Service Structure tree. If set then the contents of the tree are not automatically refreshed on tree node expansion.
Refresh	Obtains an updated list of items from ORBACUS NOTIFY and updates the console display accordingly. This option is useful if the list of items has been changed by another ORBACUS NOTIFY client.

5.2.5 The Help Menu

This menu is used to access the on-line help facilities.

Help Contents	Displays the main help contents page. From here the user can navigate the entire on-line help system.
About...	Displays version and copyright information.

5.3 The Toolbar

The ORBACUS NOTIFY Console toolbar is shown in 5.2. It contains buttons for the most commonly used menus commands.

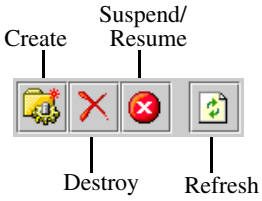

Figure 5.2: ORBACUS NOTIFY **Console Toolbar**

5.4 *The Popup Menu*

Right-clicking on the various items in the console displays a context sensitive popup menu, as shown in 5.3. This popup menu is a shortcut to the menu commands and contains

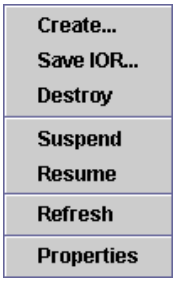

Figure 5.3: Popup Menu

appropriate operations for the selected object (channel, admin, proxy or filter) based on its current state.

5.5 *Creation Wizards*

The ORBACUS NOTIFY Console guides users through the creation of various items through the use of object creation wizards. A sample wizard dialog is shown in 5.4. The initial wizard dialog is displayed by invoking the **Create...** operation on a selected object. The wizards provide instructions related to the setup of various objects within ORBACUS NOTIFY.

Figure 5.4: Sample Creation Wizard

5.6 Managing Notification Channels

5.6.1 Creating a New Channel

To create a new channel simply choose the **EventChannelFactory** and select the **Edit/ Create...** operation. The **Event Channel Creation Wizard** then steps through the creation of the channel.

5.6.2 Notification Channel Properties

The **Edit/Properties** menu operation for a selected channel displays a tabbed property dialog in which various channel properties may be edited. All the properties in this dialog are set initially when the channel is created with the channel creation wizard.

QoS Properties

The channel **QoS Properties** tab in the **Event Channel Properties** dialog is shown in 5.5. This includes all QoS properties available for the channel including ORBACUS NOTIFY

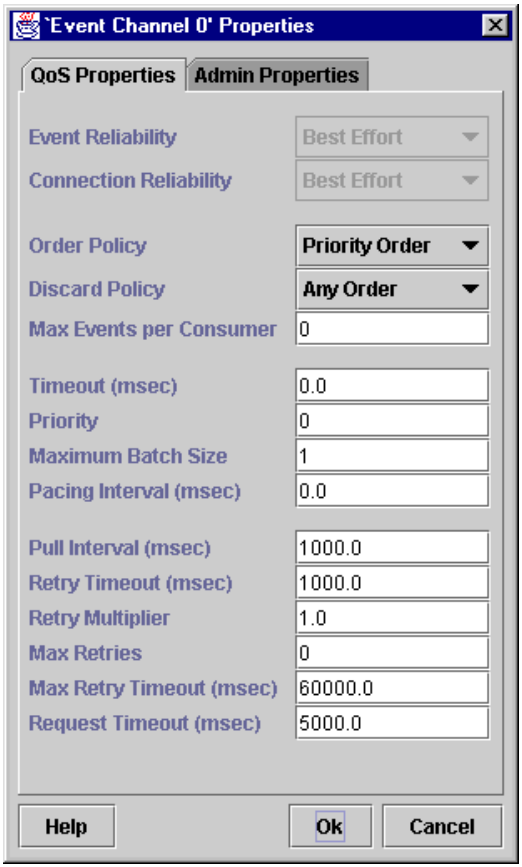

Figure 5.5: Notification Channel QoS Properties

proprietary properties. Note that **Event Reliability** and **Connection Reliability** are only set during channel creation and cannot be altered afterwards.

Admin Properties

The **Admin Properties** tab in the **Event Channel Properties** dialog, shown in 5.6, is used

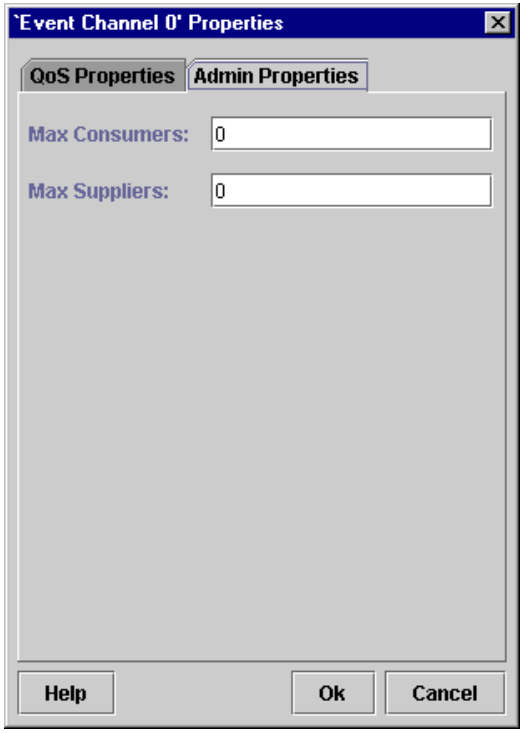

Figure 5.6: Notification Channel Admin Properties

to set the maximum number of suppliers and consumers permitted per channel.

Subscription Types

Note: The **Subscription Types** *panel is read only and is only available as a tabbed panel in the* **Object Properties** *section of the main window.*

The subscription types panel for a channel displays an aggregate list of all event types subscribed to by descendent consumer admin and proxy supplier objects.

Offered Types

Note: The **Offered Types** *panel is read only and is only available as a tabbed panel in the* **Object Properties** *section of the main window.*

The offered types panel for a channel displays an aggregate list of all event types offered by descendent supplier admin and proxy consumer objects.

5.6.3 Destroying a Channel

To destroy a channel simply select the channel and select the **Edit/Destroy** menu operation. A confirmation is displayed before the channel is removed. Note that destroying a channel also destroys all admins and proxies associated with that channel.

5.7 Managing Admins

5.7.1 Creating a New Admin

To create a new supplier or consumer admin choose **Edit/Create** on a selected event channel. The **Admin Creation Wizard** then steps through the configuration of the new admin object.

5.7.2 Admin Properties

QoS Properties

Supplier and consumer admin QoS properties are configured in **QoS Properties** tab of the **Admin Properties** dialog, shown in 5.7. This dialog is activated from the **Edit/Properties** operation when an admin object is selected in the **Service Structure** tree.

Mapping Filters

Priority and lifetime mapping filters may be assigned to or removed from consumer admin objects in the **Mapping Filters** tab of the **Admin Properties** dialog (5.8).

Subscription/Offered Types

For consumer admins, subscription event types are managed in the **Subscription Types** tab of the **Admin Properties** dialog (5.9). Similarly, for supplier admins offered event types are managed in the **Offered Types** tab of the **Admin Properties** dialog. Note that

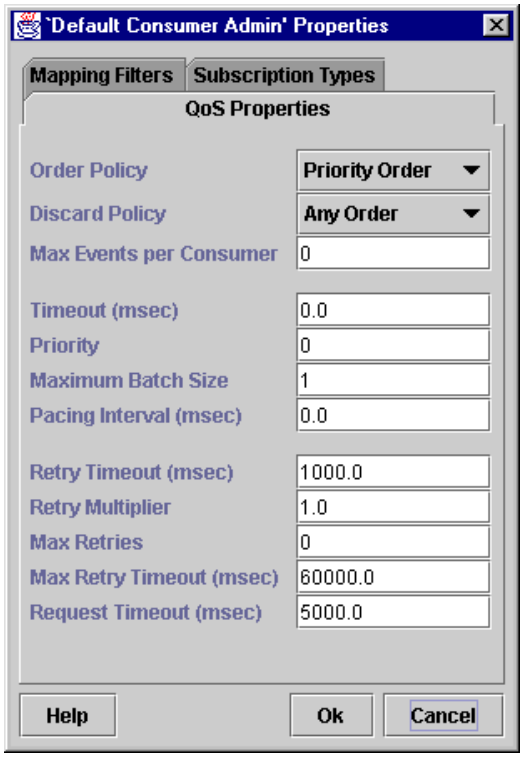

Figure 5.7: Admin QoS Properties

only one of the **Subscription Types** or **Offered Types** tab is available depending on whether a consumer or supplier admin is selected from the **Service Structure Tree**.

5.7.3 Destroying an Admin

A selected admin is destroyed by choosing the **Edit/Destroy** menu operation. A confirmation is displayed before the admin is removed. Note that destroying an admin also destroys all proxies associated with it. Any filters created from the selected admin are not destroyed. Rather the destroyed admin is removed from the filter's subscriber list.

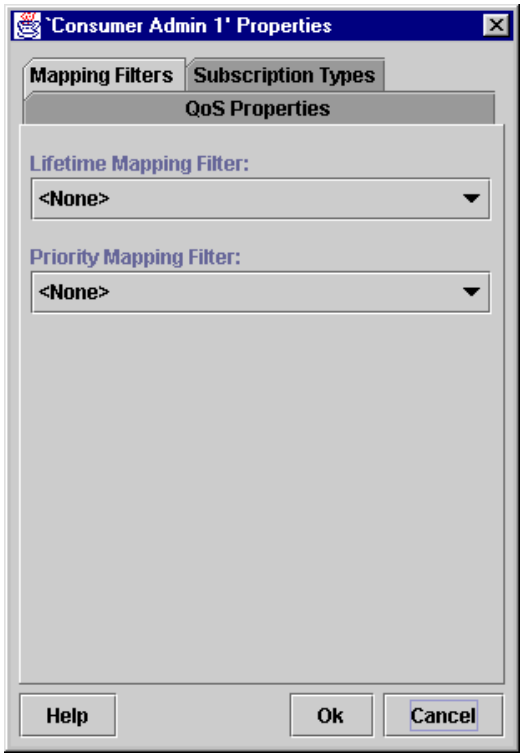

Figure 5.8: Consumer Admin Mapping Filters

5.8 Managing Proxies

5.8.1 Creating a New Proxy

Supplier and consumer proxies are created from an admin object. Supplier admins control the creation of consumer proxies while consumer admins provide methods for the creation of supplier proxies. In either case, to create a proxy from the console choose the appropriate admin and select **Edit/Create...**. The **Filter/Proxy Creation Wizard** then steps through the creation of the proxy.

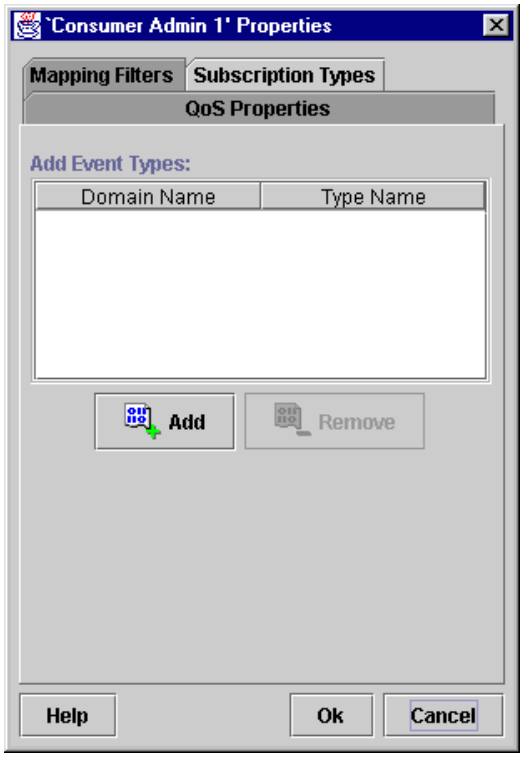

Figure 5.9: Admin Subscription/Offer Types

5.8.2 Proxy Properties

QoS Properties

Proxy QoS properties are configured in the **Proxy Properties** dialog, displayed in 5.10. Choose a proxy from the **Service Structure** tree and select **Edit/Properties** to display this dialog.

Mapping Filters

Priority and lifetime mapping filters may be assigned to or removed from supplier proxy objects in the **Mapping Filters** tab of the **Proxy Properties** dialog (5.11).

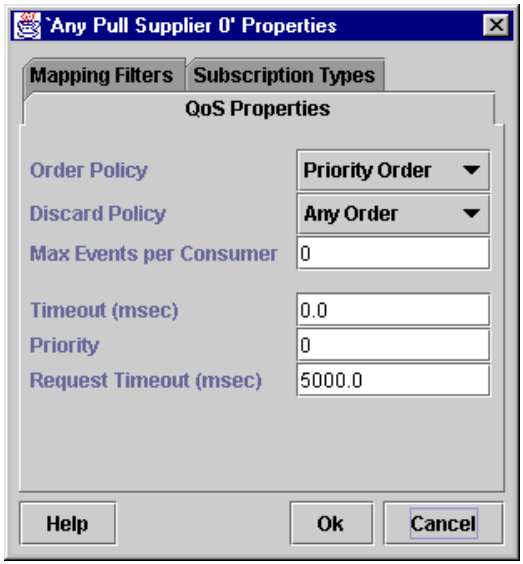

Figure 5.10: Proxy QoS Properties

Subscription/Offer Types

For supplier proxies, subscription event types are managed in the **Subscription Types** tab of the **Proxy Properties** dialog (5.12). Similarly, for consumer proxies, offered event types are managed in the **Offered Types** tab of the **Proxy Properties** dialog. Note that only one of the **Subscription Types** or **Offered Types** tab is available depending on whether a consumer or supplier proxy is selected from the **Service Structure Tree**.

Proxy Information

Note: The **Proxy Information** *panel is read only and is only available as a tabbed panel in the* **Object Properties** *section of the main window.*

The **Proxy Information** panel displays various operational properties for the selected proxy. These include:

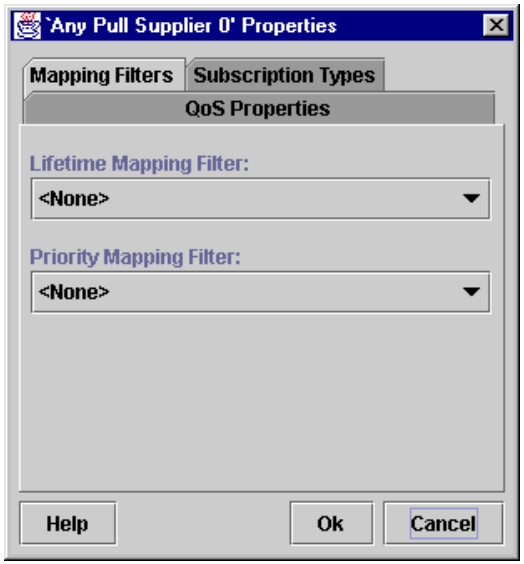

Figure 5.11: Supplier Proxy Mapping Filters

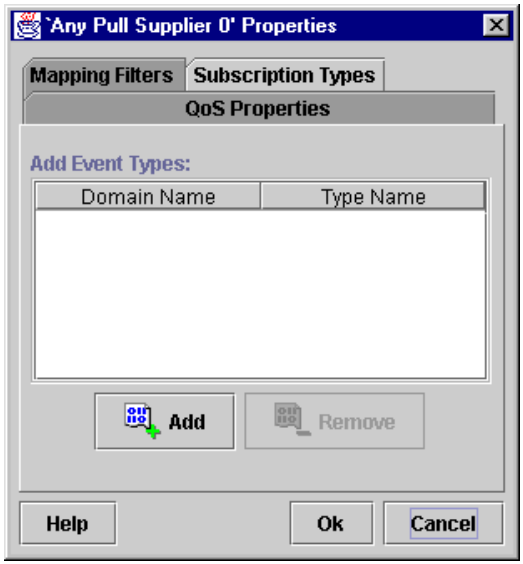

Figure 5.12: Proxy Subscription/Offer Types

Queue Size	The number of events currently queued for delivery.
Total Retries	The total number of retransmission attempts for all events handled by the target proxy.
Current Retries	The current number of retransmission attempts for the event at the head of the queue.
Discards	The number of events discarded, due to either a timeout or retransmission failure.
Event Counter	The total number of events successfully delivered.
Proxy State	The connection state of the proxy, either connected or disconnected.

Note that with the exception of **Queue Size**, all the above properties are transient. That is they are not stored persistently.

5.8.3 Destroying a Proxy

Like channels and admins, a selected proxy is destroyed by choosing the **Edit/Destroy** menu operation. A confirmation is displayed before the proxy is removed. Any filters created from the selected proxy are not destroyed. Instead the destroyed proxy is removed from the filter's subscriber list.

5.9 Managing Filters

5.9.1 Creating a New Filter

Filters can be created from any of the following objects:

- admin
- proxy
- FilterFactory

Once an object matching one of the above types is selected, invoke the **Edit/Create...** menu operation. The **Filter Creation Wizard**[1] then steps through the creation of the filter. Note that all filters become property of the **FilterFactory** and have associations with zero, one, or many admins and/or proxies.

5.9.2 Filter Properties

There are no editable properties associated with a filter. When a filter is selected in the **Service Structure Tree** the right-hand panel displays the read-only list of subscribers.

5.9.3 Destroying a Filter

A selected filter is destroyed by choosing the **Edit/Destroy** menu operation. A confirmation is displayed before the filter is removed.

1. If the filter is created from an admin then the **Filter/Proxy Creation Wizard** is used.

5.10 Managing Filter Constraints

5.10.1 Creating a New Filter Constraint

A filter constraint is created from a filter object. To create a filter constraint from the console, choose the appropriate filter and select **Edit/Create...**. The **Constraint Creation Wizard** then steps through the creation of the filter constraint.

5.10.2 Filter Constraint Properties

The **Edit/Properties** menu operation for a selected filter constraint displays a tabbed property dialog in which various constraint properties may be edited. All the properties in this dialog are set initially when the constraint is created with the **Constraint Creation Wizard**.

Expression Properties

The constraint expression is accessed with the **Expression Properties** tab, shown in Figure 5.13. This dialog supports in-place editing of the constraint expression. Constraints which do not conform to the constraint grammar cannot be entered.

Event Type Properties

The list of event types for a constraint is accessed with the **Event Type Properties** tab, shown in Figure 5.14. To add a new event type click the **Add** button, which adds a new, blank, event type to the list. All event types in the list may be edited in-place. To remove a selected event type click the **Remove** button.

5.10.3 Destroying a Filter Constraint

A selected filter constraint is destroyed by choosing the **Edit/Destroy** menu operation. A confirmation is displayed before the constraint is removed.

5.11 Managing Mapping Filters

5.11.1 Creating a New Mapping Filter

Mapping filters may only be created from the FilterFactory. Existing mapping filters may be assigned to the following objects from the appropriate properties dialog:

- consumer admin

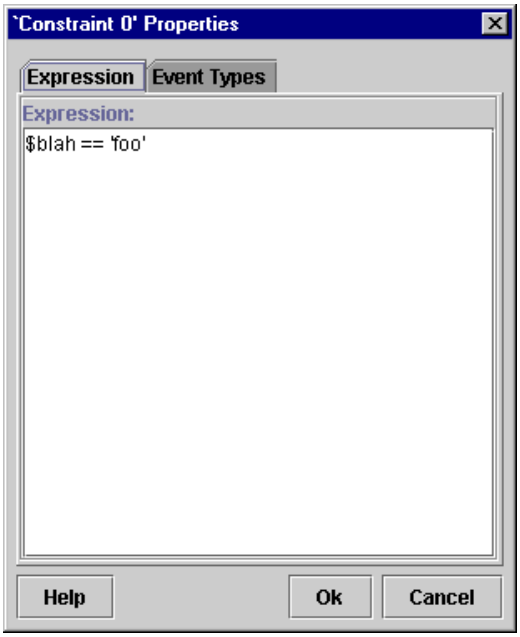

Figure 5.13: Constraint Expression Properties

- supplier proxy

To create a new mapping filter, select the FilterFactory and invoke the **Edit/Create...** menu operation. The **Filter Creation Wizard**[1] then steps through the creation of the filter. Note that all mapping filters are property of the **FilterFactory** and have associations with zero, one, or many admins and/or proxies.

5.11.2 Mapping Filter Properties

There are no editable properties associated with a mapping filter. When a filter is selected in the **Service Structure Tree** the right-hand panel displays the read-only list of subscribers and the default value associated with the mapping filter.

1. If the filter is created from a consumer admin then the **Filter/Proxy Creation Wizard** is used.

Figure 5.14: Constraint Event Type Properties

5.11.3 Destroying a Mapping Filter

A selected mapping filter is destroyed by choosing the **Edit/Destroy** menu operation. A confirmation is displayed before the mapping filter is removed.

5.12 Managing Mapping Filter Constraint-Value Pairs

5.12.1 Creating a New Constraint-Value Pair

A constraint-value pair is created from a mapping filter object. To create a constraint-value pair from the console, choose the appropriate mapping filter and select **Edit/Create...**. The **Constraint Creation Wizard** then steps through the creation of the constraint-value pair.

5.12.2 Constraint-Value Pair Properties

The **Edit/Properties** menu operation for a selected mapping filter constraint-value pair displays a tabbed property dialog in which various constraint properties may be edited. All the properties in this dialog are set initially when the constraint-value pair is created with the **Constraint Creation Wizard**.

Expression Properties

The constraint expression is accessed with the **Expression Properties** tab, shown in 5.15. This dialog supports in-place editing of the constraint expression. Constraints which do not conform to the constraint grammar cannot be entered.

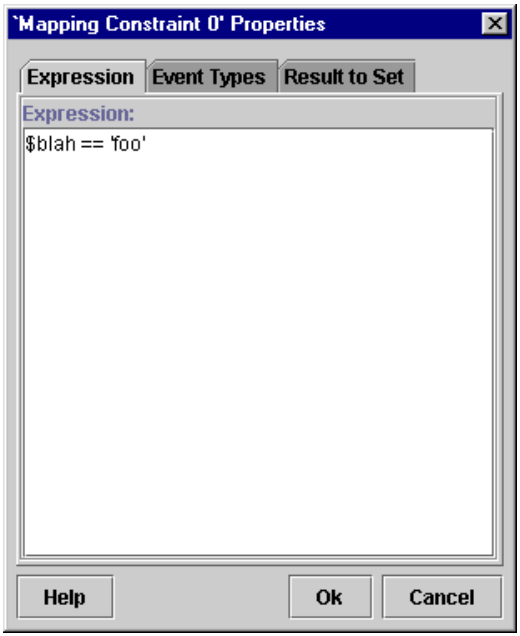

Figure 5.15: Constraint Expression Properties

Event Type Properties

The list of event types for a constraint is accessed with the **Event Type Properties** tab, shown in 5.16. To add a new event type click the **Add** button, which adds a new, blank,

event type to the list. All event types in the list may be edited in-place. To remove a selected event type click the **Remove** button.

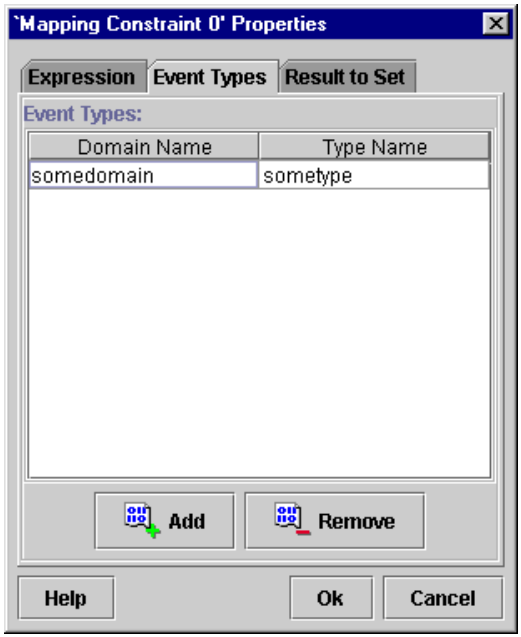

Figure 5.16: Constraint Event Type Properties

Result to Set Properties

The value to be returned by a mapping filter on a match with a constraint may be edited in the **Result to Set Properties** tab (5.17).

5.12.3 Destroying a Constraint-Value Pair

A selected mapping filter constraint-value pair is destroyed by choosing the **Edit/Destroy** menu operation. A confirmation is displayed before the constraint-value pair is removed.

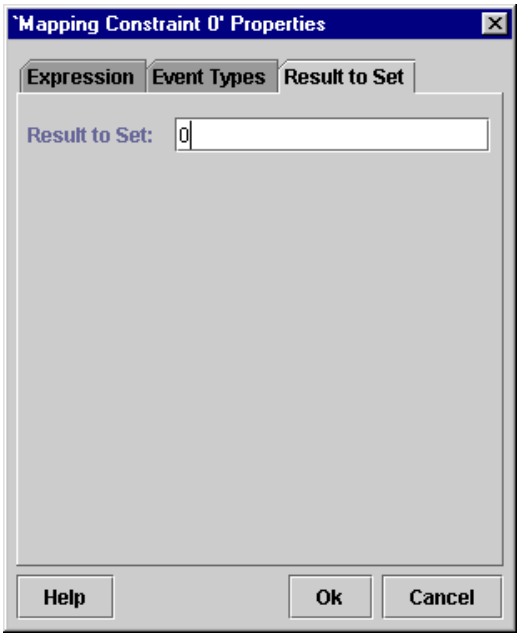

Figure 5.17: Constraint Result to Set Properties

APPENDIX A CosEventChannelAdmin Reference

A.1 Module CosEventChannelAdmin

This module contains channel administration interfaces. These interfaces support the creation of the various Event Service type admin and proxy objects.

Exceptions

AlreadyConnected

```
exception AlreadyConnected
{
};
```

Thrown by a consumer or supplier proxy to indicate that a client is already registered. The proxy interfaces permit only one connection at a time.

TypeError

```
exception TypeError
{
};
```

Certain proxy implementations may impose additional requirements on pull suppliers and push consumers that are allowed to connect. If the object does not support these requirements the TypeError exception is raised.

A.2 Interface CosEventChannelAdmin::ProxyPushConsumer

interface **ProxyPushConsumer**
inherits from CosEventComm::PushConsumer

A push supplier uses this interface to register with an event channel.

Operations

connect_push_supplier

```
void connect_push_supplier(in CosEventComm::PushSupplier push_supplier)
   raises(AlreadyConnected);
```

Registers a push supplier implementation with the event channel. A push supplier need not implement a `CosEventComm::PushSupplier` object to successfully push events on the channel. This is only necessary if the supplier wishes for notification when it is disconnected by the channel. If this notification is not required a nil object reference may be given.

Parameters:

`push_supplier` - A reference to a push supplier implementation, or a nil object reference.

A.3 Interface CosEventChannelAdmin::ProxyPullSupplier

interface **ProxyPullSupplier**
inherits from CosEventComm::PullSupplier

A pull consumer uses this interface to register with an event channel.

Operations

connect_pull_consumer

```
void connect_pull_consumer(in CosEventComm::PullConsumer pull_consumer)
   raises(AlreadyConnected);
```

Registers a pull consumer implementation with the event channel. A pull consumer need not implement a `CosEventComm::PullConsumer` object to successfully pull events from a channel. This is only necessary if the consumer wishes for notification when it is disconnected by the channel. If this notification is not required a nil object reference may be passed.

Parameters:

`pull_consumer` - A reference to a pull consumer implementation, or a nil object reference.

A.4 Interface CosEventChannelAdmin::ProxyPullConsumer

interface **ProxyPullConsumer**
inherits from CosEventComm::PullConsumer

A pull supplier uses this interface to register with an event channel.

Operations

connect_pull_supplier

```
void connect_pull_supplier(in CosEventComm::PullSupplier pull_supplier)
   raises(AlreadyConnected,
          TypeError);
```

Registers a pull supplier implementation with the event channel. A pull supplier must implement and register a `CosEventComm::PullSupplier` object so that the channel may successfully pull events from it.

Parameters:

`pull_supplier` - A reference to a pull supplier implementation,

A.5 Interface CosEventChannelAdmin::ProxyPushSupplier

interface **ProxyPushSupplier**
inherits from CosEventComm::PushSupplier

A push consumer uses this interface to register with an event channel.

Operations

connect_push_consumer

```
void connect_push_consumer(in CosEventComm::PushConsumer push_consumer)
   raises(AlreadyConnected,
          TypeError);
```

Registers a push consumer implementation with the event channel. A push consumer must implement and register a `CosEventComm::PushConsumer` object so that the channel may successfully push events on it.

Parameters:

`push_consumer` - A reference to a push consumer implementation,

A.6 Interface CosEventChannelAdmin::ConsumerAdmin

interface **ConsumerAdmin**

An event consumer uses this interface to create the appropriate proxy supplier.

Operations

obtain_push_supplier

```
ProxyPushSupplier obtain_push_supplier();
```

Creates a new `ProxyPushSupplier` object.

Returns:
An object reference to the new proxy is returned.

obtain_pull_supplier

```
ProxyPullSupplier obtain_pull_supplier();
```

Creates a new `ProxyPullSupplier` object.

Returns:
An object reference to the new proxy is returned.

A.7 Interface CosEventChannelAdmin::SupplierAdmin

interface **SupplierAdmin**

An event supplier uses this interface to create the appropriate proxy consumer.

Operations

obtain_push_consumer

```
ProxyPushConsumer obtain_push_consumer();
```

Creates a new `ProxyPushConsumer` object.

Returns:
An object reference to the new proxy is returned.

obtain_pull_consumer

```
ProxyPullConsumer obtain_pull_consumer();
```

Creates a new `ProxyPullConsumer` object.

Returns:
An object reference to the new proxy is returned.

A.8 Interface CosEventChannelAdmin::EventChannel

interface **EventChannel**

Event suppliers and consumers use the `EventChannel` interface to obtain the admin objects required for proxy creation.

Operations

for_consumers

```
ConsumerAdmin for_consumers();
```

Creates a new `ConsumerAdmin` object.

Returns:
An object reference to the new admin is returned.

for_suppliers

```
SupplierAdmin for_suppliers();
```

Creates a new `SupplierAdmin` object.

Returns:
An object reference to the new admin is returned.

destroy

```
void destroy();
```

Destroys an `EventChannel` and all associated admin and proxy objects.

APPENDIX B CosEventComm Reference

B.1 Module CosEventComm

This module contains the basic, Event Service compatible, interfaces supporting the exchange of events between a supplier and consumer. Note that a channel acts as both supplier and consumer of events through its proxy interfaces.

Exceptions

Disconnected

```
exception Disconnected
{
};
```

This exception is raised by an operation if event communication has been disconnected.

B.2 Interface CosEventComm::PushConsumer

interface **PushConsumer**

This interface is implemented by a push consumer to receive event data.

Operations

push

```
void push(in any data)
   raises(Disconnected);
```

A supplier invokes the `push` operation to transfer an event to a consumer.

Parameters:

`data` - The event is encapsulated in a `CORBA::Any`.

disconnect_push_consumer

```
void disconnect_push_consumer();
```

This method terminates event communication and releases resources allocated by the target object.

B.3 Interface CosEventComm::PushSupplier

interface **PushSupplier**

This interface is implemented by a push supplier which wishes to receive notification when it is disconnected.

Operations

disconnect_push_supplier

```
void disconnect_push_supplier();
```

This method terminates event communication and releases resources allocated by the target object.

B.4 Interface CosEventComm::PullSupplier

interface **PullSupplier**

This interface is implemented by a pull supplier so that the channel my pull events.

Operations

pull

```
any pull()
   raises(Disconnected);
```

This method blocks the calling thread until the supplier has data available or an exception is raised.

Returns:
An event in a `CORBA::Any`.

try_pull

```
any try_pull(out boolean has_event)
   raises(Disconnected);
```

This method does not block and can be used to poll a pull supplier for events.

Parameters:
`has_event` - Set to `TRUE` if there is an event available, `FALSE` otherwise.

Returns:
An event in a `CORBA::Any` if `has_event` is `TRUE`, undefined if `has_event` is `FALSE`.

disconnect_pull_supplier

```
void disconnect_pull_supplier();
```

This method terminates event communication and releases resources allocated by the target object.

B.5 Interface CosEventComm::PullConsumer

interface **PullConsumer**

This interface is implemented by a pull consumer which wishes to receive notification when it is disconnected.

Operations

disconnect_pull_consumer

```
void disconnect_pull_consumer();
```

This method terminates event communication and releases resources allocated by the target object.

APPENDIX C CosNotification Reference

C.1 Module CosNotification

This module contains the definition of the structured event type and various definitions related to QoS and Administration properties.

Aliases

Istring

```
typedef string Istring;
```

PropertyName

```
typedef Istring PropertyName;
```

Alias for a property name.

PropertyValue

```
typedef any PropertyValue;
```

Alias for a property value.

PropertySeq

```
typedef sequence<Property> PropertySeq;
```

Alias for a sequence of property name-value pairs.

OptionalHeaderFields

```
typedef PropertySeq OptionalHeaderFields;
```

Alias for event header optional header fields.

FilterableEventBody

```
typedef PropertySeq FilterableEventBody;
```

Alias for event body filterable fields.

QoSProperties

```
typedef PropertySeq QoSProperties;
```

Alias for Quality of Service properties.

AdminProperties

```
typedef PropertySeq AdminProperties;
```

Alias for channel administration properties.

EventTypeSeq

```
typedef sequence<EventType> EventTypeSeq;
```

Alias for a sequence of event types.

NamedPropertyRangeSeq

```
typedef sequence<NamedPropertyRange> NamedPropertyRangeSeq;
```

Alias for a sequence of named property ranges.

PropertyErrorSeq

```
typedef sequence<PropertyError> PropertyErrorSeq;
```

Alias for a sequence of property errors.

EventBatch

```
typedef sequence<StructuredEvent> EventBatch;
```

Alias for a sequence of structured events.

Constants

EventReliability

```
const string EventReliability = "EventReliability";
```

Specifies event reliability. The valid values are `BestEffort` and `Persistent`.

BestEffort

```
const short BestEffort = 0;
```

Reliability property value.

Persistent

```
const short Persistent = 1;
```

Reliability property value.

ConnectionReliability

```
const string ConnectionReliability = "ConnectionReliability";
```

Specifies connection reliability. The valid values are `BestEffort` and `Persistent`.

Priority

```
const string Priority = "Priority";
```

Indicates the relative priority of the event compared to other events in the channel. Can take on any value between -32,767 and 32,767, with -32,767 being the lowest priority, 32,767 being the highest, and 0 being the default.

LowestPriority

```
const short LowestPriority = -32767;
```

Priority property value.

HighestPriority

```
const short HighestPriority = 32767;
```

Priority property value.

DefaultPriority

```
const short DefaultPriority = 0;
```

Priority property value.

StartTime

```
const string StartTime = "StartTime";
```

Gives an absolute time (e.g., 12/12/99 at 23:59) after which the channel may deliver the event. The value for this property is of type `TimeBase:UtcT`.

StopTime

```
const string StopTime = "StopTime";
```

Gives an absolute time (e.g., 12/12/99 at 23:59) at which the channel should discard the event. The value for this property is of type `TimeBase:UtcT`.

Timeout

```
const string Timeout = "Timeout";
```

Gives a relative time (e.g., 10 minutes from time received) after which the channel should discard the event. The value 0 indicates there is no timeout. The value for this property is of type `TimeBase:TimeT`.

OrderPolicy

```
const string OrderPolicy = "OrderPolicy";
```

This QoS property sets the policy used by a given proxy to order the events it has buffered for delivery (either to another proxy or a consumer). Constant values to represent the permitted settings are defined.

AnyOrder

```
const short AnyOrder = 0;
```

`OrderPolicy` property value indicating any ordering policy is permitted.

FifoOrder

```
const short FifoOrder = 1;
```

`OrderPolicy` property value indicating events should be delivered in the order of their arrival.

PriorityOrder

```
const short PriorityOrder = 2;
```

`OrderPolicy` property value indicating events should be buffered in priority order, such that higher priority events will be delivered before lower priority events.

DeadlineOrder

```
const short DeadlineOrder = 3;
```

`OrderPolicy` property value indicating events should be buffered in the order of shortest expiry deadline first, such that events that are destined to timeout soonest should be delivered first.

DiscardPolicy

```
const string DiscardPolicy = "DiscardPolicy";
```

Discard policy determines the order in which events are discarded when the number of queued events exceeds `MaxEventsPerConsumer`. The `OrderPolicy` property values are also `DiscardPolicy` property values.

LifoOrder

```
const short LifoOrder = 4;
```

`DiscardPolicy` property value. The last event received will be the first discarded.

RejectNewEvents

```
const short RejectNewEvents = 5;
```

`DiscardPolicy` property value. The proxy consumers of the associated channel should reject attempts to send new events to the channel when such an attempt would result in a buffer overflow, raising the system exception `IMPL_LIMIT`. Note that this is the default setting for discard policy.

MaximumBatchSize

```
const string MaximumBatchSize = "MaximumBatchSize";
```

This QoS property has meaning in the case of consumers that register to receive sequences of structured events. For any such consumer, this property indicates the maximum number of events that will be delivered within each sequence. The corresponding value is of type `long`.

PacingInterval

```
const string PacingInterval = "PacingInterval";
```

This QoS property has meaning in the case of consumers that register to receive sequences of structured events. For any such consumer, this property defines the maximum period of time the channel will collect individual events into a sequence before delivering the sequence to the consumer. The corresponding value is of type `TimeBase::TimeT`.

StartTimeSupported

```
const string StartTimeSupported = "StartTimeSupported";
```

Indicates whether or not the setting of `StartTime` on a per-message basis is supported. The corresponding value is of type `boolean`.

StopTimeSupported

```
const string StopTimeSupported = "StopTimeSupported";
```

Indicates whether or not the setting of `StopTime` on a per-message basis is supported. The corresponding value is of type `boolean`.

MaxEventsPerConsumer

```
const string MaxEventsPerConsumer = "MaxEventsPerConsumer";
```

An administrative property can be set on a channel to bound the maximum number of events a given channel is allowed to queue at any given point in time. However, a single badly behaved consumer could result in the channel holding the maximum number of events it is allowed to queue for an extended period of time, preventing further event communication through the channel. This QoS property helps to avoid this situation by bounding the maximum number of events the channel will queue on behalf of a given consumer. The corresponding value is of type `long`.

MaxQueueLength

```
const string MaxQueueLength = "MaxQueueLength";
```

The maximum number of events that a channel will buffer at any one time. The corresponding value is of type `long`.

MaxConsumers

```
const string MaxConsumers = "MaxConsumers";
```

The maximum number of consumers that can be connected to a channel at any one time. The corresponding value is of type `long`.

MaxSuppliers

```
const string MaxSuppliers = "MaxSuppliers";
```

The maximum number of suppliers that can be connected to a channel at any one time. The corresponding value is of type `long`.

Structs

Property

```
struct Property
{
   PropertyName name;
   PropertyValue value;
};
```

A generic name-value property pair.

Members:

`name` - The name of the property.
`value` - The value of the property.

EventType

```
struct EventType
{
   string domain_name;
   string type_name;
};
```

Structure defining an event type. The type of an event is governed by the `domain_name` and `type_name`.

Members:

`domain_name` - Identifies the vertical industry domain in which the event is defined.
`type_name` - Further classifies the event within the domain.

PropertyRange

```
struct PropertyRange
{
   PropertyValue low_val;
   PropertyValue high_val;
};
```

Structure used to indicate a range of acceptable values for an unnamed property.

NamedPropertyRange

```
struct NamedPropertyRange
```

```
{
   PropertyName name;
   PropertyRange range;
};
```

Structure used to indicate a range of acceptable values for a named property.

PropertyError

```
struct PropertyError
{
   QoSError_code code;
   PropertyName name;
   PropertyRange available_range;
};
```

Structure to indicate a property error for the name property and, if applicable, a suitable range of values.

FixedEventHeader

```
struct FixedEventHeader
{
   EventType event_type;
   string event_name;
};
```

Structured event fixed header

Members:

`event_type` - Categorizes the event.
`event_name` - A name given to this event instance to differentiate it from other events of the same type.

EventHeader

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

Structured event header

Members:

`fixed_header` - Categorizes and names the event.
`variable_header` - Optional header information. This may contain any name-value pair that the user chooses. Standard values are related to per event QoS settings.

StructuredEvent

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

The StructuredEvent Type. Events transmitted in this form are subject to filtering.

Exceptions

UnsupportedQoS

```
exception UnsupportedQoS
{
   PropertyErrorSeq qos_err;
};
```

This exception is raised when a channel or channel component cannot satisfy a client's QoS request.

Members:

`qos_err` - Contains a list of the rejected QoS settings, along with reason for rejection, and suitable property values, if applicable.

UnsupportedAdmin

```
exception UnsupportedAdmin
{
   PropertyErrorSeq admin_err;
};
```

This exception is raised when a channel or proxy does not support the requested aministrative property settings.

Members:

`admin_err` - Contains a list of the rejected administrative settings, along with reason for rejection, and suitable property values, if applicable.

Enums

QoSError_code

```
enum QoSError_code
```

```
{
   UNSUPPORTED_PROPERTY,
   UNAVAILABLE_PROPERTY,
   UNSUPPORTED_VALUE,
   UNAVAILABLE_VALUE,
   BAD_PROPERTY,
   BAD_TYPE,
   BAD_VALUE
};
```

Error codes used to indicate an invalid property assignment.

Members:

`UNSUPPORTED_PROPERTY` - Property not supported by this implementation of the target object.

`UNAVAILABLE_PROPERTY` - Property cannot be set within the current context of other property settings.

`UNSUPPORTED_VALUE` - The property value is not supported by this implementation of the target object.

`UNAVAILABLE_VALUE` - The property value is not supported within the current context of other property settings.

`BAD_PROPERTY` - Unrecognized property name.

`BAD_TYPE` - Incorrect value type for this property.

`BAD_VALUE` - Illegal value for this property.

C.2 Interface CosNotification::QoSAdmin

interface **QoSAdmin**

Supports the management of QoS property settings.

Operations

get_qos

```
QoSProperties get_qos();
```

Retrieves the current list of QoS properties for the target object.

Returns:
A sequence of QoS property name-value pairs.

set_qos

```
void set_qos(in QoSProperties qos)
   raises(UnsupportedQoS);
```

Incrementally applies QoS settings to the target object. New elements are appended to the list of QoS properties already associated with the target object. If the property already exists for the target object its value is changed to the new setting.

Parameters:
`qos` - A list of QoS properties.

validate_qos

```
void validate_qos(in QoSProperties required_qos,
                  out NamedPropertyRangeSeq available_qos)
   raises(UnsupportedQoS);
```

Checks to see if a list of QoS properties are supported by the target object without changing the list of properties already associated with the object. If any of the properties in `required_qos` are not supported the `UnsupportedQoS` exception is raised.

Parameters:
`required_qos` - The QoS properties of interest to the caller are passed in this parameter.
`available_qos` - If the properties in `required_qos` are supported, other optional QoS properties which are also supported are returned in this parameter.

C.3 Interface CosNotification::AdminPropertiesAdmin

interface **AdminPropertiesAdmin**

Supports the management of administrative properties.

Operations

get_admin

```
AdminProperties get_admin();
```

Retrieves the list of administrative properties associated with the target object.

Returns:
A sequence of admin name-value pairs.

set_admin

```
void set_admin(in AdminProperties admin)
   raises(UnsupportedAdmin);
```

Sets the administrative properties for the target object. If any of the properties in `admin` are unsupported, the `UnsupportedAdmin` exception is raised.

Parameters:
`admin` - A sequence of name-value pairs defining the administrative properties to be set on the target object.

APPENDIX D CosNotifyChannelAdmin Reference

D.1 Module CosNotifyChannelAdmin

This module contains the definitions of the primary Notification Service interfaces. These interfaces allow suppliers and consumers to connect to a channel.

Aliases

ProxyID

```
typedef long ProxyID;
```

Alias for a proxy ID.

ProxyIDSeq

```
typedef sequence<ProxyID> ProxyIDSeq;
```

Alias for a sequence of Proxy IDs.

AdminID

```
typedef long AdminID;
```

Alias for an admin ID.

AdminIDSeq

```
typedef sequence<AdminID> AdminIDSeq;
```

Alias for a sequence of Admin IDs.

ChannelID

```
typedef long ChannelID;
```

Alias for a channel ID.

ChannelIDSeq

```
typedef sequence<ChannelID> ChannelIDSeq;
```

Alias for a sequence of channel IDs.

Structs

AdminLimit

```
struct AdminLimit
{
   CosNotification::PropertyName name;
   CosNotification::PropertyValue value;
};
```

Contains a property name-value pair representing a limit on the number of proxies that may connected to an admin object.

Exceptions

ConnectionAlreadyActive

```
exception ConnectionAlreadyActive
{
};
```

Raised on an attempt to resume an already active connection.

ConnectionAlreadyInactive

```
exception ConnectionAlreadyInactive
{
};
```

Raised on an attempt to suspend an already inactive connection.

NotConnected

```
exception NotConnected
{
};
```

Raised on an attempt to suspend or a resume a disconnected proxy.

AdminNotFound

```
exception AdminNotFound
{
};
```

Raised when an admin identified by an `AdminID` cannot be found.

ProxyNotFound

```
exception ProxyNotFound
{
};
```

Raised when a proxy identified by a `ProxyID` cannot be found.

AdminLimitExceeded

```
exception AdminLimitExceeded
{
   AdminLimit admin_property_err;
};
```

Raised on an attempt to connect a proxy which would exceed the maximum number allowed for the target admin object.

ChannelNotFound

```
exception ChannelNotFound
{
};
```

Indicates that a channel with a given channel ID was not found.

Enums

ProxyType

```
enum ProxyType
{
   PUSH_ANY,
   PULL_ANY,
```

```
   PUSH_STRUCTURED,
   PULL_STRUCTURED,
   PUSH_SEQUENCE,
   PULL_SEQUENCE
};
```

Supplier and consumer proxy types.

Members:

`PUSH_ANY` - Push delivery model, any events.
`PULL_ANY` - Pull delivery model, any events.
`PUSH_STRUCTURED` - Push delivery model, structured events.
`PULL_STRUCTURED` - Pull delivery model, structured events.
`PUSH_SEQUENCE` - Push delivery model, sequence of structured events.
`PULL_SEQUENCE` - Pull delivery model, sequence of structured events.

ObtainInfoMode

```
enum ObtainInfoMode
{
   ALL_NOW_UPDATES_OFF,
   ALL_NOW_UPDATES_ON,
   NONE_NOW_UPDATES_OFF,
   NONE_NOW_UPDATES_ON
};
```

Configures the mode by which event types are communicated during subscription sharing.

Members:

`ALL_NOW_UPDATES_OFF` - Operation should return all types known by the target object and disable automatic updates.
`ALL_NOW_UPDATES_ON` - Operation should return all types known by the target object and enable automatic updates.
`NONE_NOW_UPDATES_OFF` - Operation should disable automatic updates and return no event types.
`NONE_NOW_UPDATES_ON` - Operation should enable aautomatic updates and return no event types.

ClientType

```
enum ClientType
{
   ANY_EVENT,
   STRUCTURED_EVENT,
   SEQUENCE_EVENT
};
```

Notification Service client types, based on supported event type.

Members:

`ANY_EVENT` - Supports unstructured event delivery.
`STRUCTURED_EVENT` - Supports structured event delivery.
`SEQUENCE_EVENT` - Supports sequences of structured events.

InterFilterGroupOperator

```
enum InterFilterGroupOperator
{
   AND_OP,
   OR_OP
};
```

The `InterFilterGroupOperator` determines how filter results from an admin object and its child proxy object are combined.

Members:

`AND_OP` - Use logical AND semantics between admin and proxy filter results.
`OR_OP` - Use logical OR semantics between admin and proxy filter results.

D.2 Interface CosNotifyChannelAdmin::ProxyConsumer

interface **ProxyConsumer**
inherits from CosNotification::QoSAdmin, CosNotifyFilter::FilterAdmin

`ProxyConsumer` interface. supports operations common to all proxy consumers.

Attributes

MyType

```
readonly attribute ProxyType MyType;
```

The type (delivery model and event type) of the proxy.

MyAdmin

```
readonly attribute SupplierAdmin MyAdmin;
```

Reference to the parent supplier admin object.

Operations

obtain_subscription_types

```
CosNotification::EventTypeSeq obtain_subscription_types(in ObtainInfo-
Mode mode);
```

Obtains an aggregate list of all event types on the channel to which there is a subscription.

Parameters:
`mode` - Determines how subscribed event types are returned.

Returns:
A sequence of event types representing all events currently subscribed to on the channel.

validate_event_qos

```
void validate_event_qos(in CosNotification::QoSProperties required_qos,
                        out CosNotification::NamedPropertyRangeSeq
available_qos)
   raises(CosNotification::UnsupportedQoS);
```

Checks for a conflict between per event QoS and the QoS settings of the target proxy. If the target proxy cannot honor any of QoS properties in `required_qos` an `UnsupportedQoS` excep-

tion is raised.

Parameters:

`required_qos` - The QoS properties of interest to the caller are passed in this parameter.
`available_qos` - If the properties in `required_qos` are supported, other optional QoS properties which are also supported are returned in this parameter.

D.3 Interface CosNotifyChannelAdmin::ProxySupplier

interface **ProxySupplier**
inherits from CosNotification::QoSAdmin, CosNotifyFilter::FilterAdmin

The ProxySupplier interface supports operations common to all proxy suppliers.

Attributes

MyType

```
readonly attribute ProxyType MyType;
```

The type (delivery model and event type) of the proxy.

MyAdmin

```
readonly attribute ConsumerAdmin MyAdmin;
```

Reference to the parent consumer admin object.

priority_filter

```
attribute CosNotifyFilter::MappingFilter priority_filter;
```

Reference to an optional priority mapping filter.

lifetime_filter

```
attribute CosNotifyFilter::MappingFilter lifetime_filter;
```

Reference to an optional lifetime mapping filter.

Operations

obtain_offered_types

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

Obtains an aggregate list of all event types currently offered on the channel.

Parameters:

`mode` - Determines how offered event types are returned.

Returns:

A sequence of event types representing all events currently offered on the channel.

validate_event_qos

```
void validate_event_qos(in CosNotification::QoSProperties required_qos,
                        out CosNotification::NamedPropertyRangeSeq
available_qos)
   raises(CosNotification::UnsupportedQoS);
```

Checks for a conflict between per event QoS and the QoS settings of the target proxy. If the target proxy cannot honor any of QoS properties in `required_qos` an `UnsupportedQoS` exception is raised.

Parameters:

`required_qos` - The QoS properties of interest to the caller are passed in this parameter.
`available_qos` - If the properties in `required_qos` are supported, other optional QoS properties which are also supported are returned in this parameter.

D.4 Interface CosNotifyChannelAdmin::ProxyPushConsumer

interface **ProxyPushConsumer**
inherits from CosNotifyChannelAdmin::ProxyConsumer, CosNotifyComm::PushConsumer

The ProxyPushConsumer interface supports connections by suppliers who wish to push unstructured (`CORBA::Any`) events.

Operations

connect_any_push_supplier

```
void connect_any_push_supplier(in CosEventComm::PushSupplier
push_supplier)
   raises(CosEventChannelAdmin::AlreadyConnected);
```

Connects a supplier to the channel. If a supplier is already connected the `AlreadyConnected` exception is raised.

Parameters:

`push_supplier` - A reference to the supplier object. A nil reference is permitted.

D.5 Interface CosNotifyChannelAdmin::StructuredProxyPushConsumer

interface **StructuredProxyPushConsumer**
inherits from CosNotifyChannelAdmin::ProxyConsumer, CosNotifyComm::StructuredPushConsumer

The `StructuredProxyPushConsumer` interface supports connections by suppliers who wish to push structured events on the channel.

Operations

connect_structured_push_supplier

```
void connect_structured_push_supplier(in CosNotifyComm::Structured-
PushSupplier push_supplier)
   raises(CosEventChannelAdmin::AlreadyConnected);
```

Connects a supplier to the channel. If a supplier is already connected the `AlreadyConnected` exception is raised.

Parameters:

`push_supplier` - A reference to the supplier object. A nil reference is permitted.

D.6 Interface CosNotifyChannelAdmin::SequenceProxyPushConsumer

interface **SequenceProxyPushConsumer**
inherits from CosNotifyChannelAdmin::ProxyConsumer, CosNotifyComm::SequencePushConsumer

The `SequenceProxyPushConsumer` interface supports connections by suppliers who wish to supply sequences of structured events to the channel.

Operations

connect_sequence_push_supplier

```
void connect_sequence_push_supplier(in CosNotifyComm::SequencePushSup-
plier push_supplier)
   raises(CosEventChannelAdmin::AlreadyConnected);
```

Connects a supplier to the channel. If a supplier is already connected the `AlreadyConnected` exception is raised.

Parameters:

`push_supplier` - A reference to the supplier object. A nil reference is permitted.

D.7 Interface CosNotifyChannelAdmin::ProxyPullSupplier

interface **ProxyPullSupplier**
inherits from CosNotifyChannelAdmin::ProxySupplier, CosNotifyComm::PullSupplier

The `ProxyPullSupplier` interface supports connections by consumers who wish to pull unstructured events from the channel.

Operations

connect_any_pull_consumer

```
void connect_any_pull_consumer(in CosEventComm::PullConsumer
pull_consumer)
   raises(CosEventChannelAdmin::AlreadyConnected);
```

Connects a consumer to the channel. If a consumer is already connected the `AlreadyConnected` exception is raised.

Parameters:

`pull_consumer` - A reference to the consumer object. A nil reference is permitted.

D.8 Interface CosNotifyChannelAdmin::StructuredProxyPullSupplier

interface **StructuredProxyPullSupplier**
inherits from CosNotifyChannelAdmin::ProxySupplier, CosNotifyComm::StructuredPullSupplier

The `StructuredProxyPullSupplier` interface supports connections by consumers who wish to pull structured events from the channel.

Operations

connect_structured_pull_consumer

```
void connect_structured_pull_consumer(in CosNotifyComm::Structured-
PullConsumer pull_consumer)
   raises(CosEventChannelAdmin::AlreadyConnected);
```

Connects a consumer to the channel. If a consumer is already connected the `AlreadyConnected` exception is raised.

Parameters:

`pull_consumer` - A reference to the consumer object. A nil reference is permitted.

D.9 Interface CosNotifyChannelAdmin::SequenceProxyPullSupplier

interface **SequenceProxyPullSupplier**
inherits from CosNotifyChannelAdmin::ProxySupplier, CosNotifyComm::SequencePullSupplier

The `SequenceProxyPullSupplier` interface supports connections from consumers who wish to pull sequences of structured events from the channel.

Operations

connect_sequence_pull_consumer

```
void connect_sequence_pull_consumer(in CosNotifyComm::SequencePullCon-
sumer pull_consumer)
   raises(CosEventChannelAdmin::AlreadyConnected);
```

Connects a consumer to the channel. If a consumer is already connected the `AlreadyConnected` exception is raised.

Parameters:

`pull_consumer` - A reference to the consumer object. A nil reference is permitted.

D.10 Interface CosNotifyChannelAdmin::ProxyPullConsumer

interface **ProxyPullConsumer**
inherits from CosNotifyChannelAdmin::ProxyConsumer, CosNotifyComm::PullConsumer

The `ProxyPullConsumer` interface manages connections from suppliers who wish to have unstructured events pull from them by the channel.

Operations

connect_any_pull_supplier

```
void connect_any_pull_supplier(in CosEventComm::PullSupplier
pull_supplier)
   raises(CosEventChannelAdmin::AlreadyConnected,
          CosEventChannelAdmin::TypeError);
```

Connects a supplier to the channel. If a supplier is already connected the `AlreadyConnected` exception is raised.

Parameters:
`pull_supplier` - A reference to the supplier object. A nil reference is not permitted.

suspend_connection

```
void suspend_connection()
   raises(ConnectionAlreadyInactive,
          NotConnected);
```

This operation causes the target object to stop pulling events from the connected supplier. If the connection is already suspended the `ConnectionAlreadyInactive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

resume_connection

```
void resume_connection()
   raises(ConnectionAlreadyActive,
          NotConnected);
```

This operation causes the target to resume pulling events from the connected supplier. If the connection is not suspended the `ConnectionAlreadyActive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

D.11 Interface CosNotifyChannelAdmin::StructuredProxyPullConsumer

interface **StructuredProxyPullConsumer**
inherits from CosNotifyChannelAdmin::ProxyConsumer, CosNotifyComm::StructuredPullConsumer

The `StructuredProxyPullConsumer` interface manages connections from suppliers who wish to have structured events pulled from them by the channel.

Operations

connect_structured_pull_supplier

```
void connect_structured_pull_supplier(in CosNotifyComm::Structured-
PullSupplier pull_supplier)
   raises(CosEventChannelAdmin::AlreadyConnected,
          CosEventChannelAdmin::TypeError);
```

Connects a supplier to the channel. If a supplier is already connected the `AlreadyConnected` exception is raised.

Parameters:

`pull_supplier` - A reference to the supplier object. A nil reference is not permitted.

suspend_connection

```
void suspend_connection()
   raises(ConnectionAlreadyInactive,
          NotConnected);
```

This operation causes the target object to stop pulling events from the connected supplier. If the connection is already suspended the `ConnectionAlreadyInactive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

resume_connection

```
void resume_connection()
   raises(ConnectionAlreadyActive,
          NotConnected);
```

This operation causes the target to resume pulling events from the connected supplier. If the connection is not suspended the `ConnectionAlreadyActive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

D.12 Interface CosNotifyChannelAdmin::SequenceProxyPullConsumer

interface **SequenceProxyPullConsumer**
inherits from CosNotifyChannelAdmin::ProxyConsumer, CosNotifyComm::SequencePullConsumer

The `SequenceProxyPullConsumer` interface manages connections from suppliers who wish to have sequences of structured events pulled from them by the channel.

Operations

connect_sequence_pull_supplier

```
void connect_sequence_pull_supplier(in CosNotifyComm::SequencePullSup-
plier pull_supplier)
   raises(CosEventChannelAdmin::AlreadyConnected,
          CosEventChannelAdmin::TypeError);
```

Connects a supplier to the channel. If a supplier is already connected the `AlreadyConnected` exception is raised.

Parameters:

`pull_supplier` - A reference to the supplier object. A nil reference is not permitted.

suspend_connection

```
void suspend_connection()
   raises(ConnectionAlreadyInactive,
          NotConnected);
```

This operation causes the target object to stop pulling events from the connected supplier. If the connection is already suspended the `ConnectionAlreadyInactive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

resume_connection

```
void resume_connection()
   raises(ConnectionAlreadyActive,
          NotConnected);
```

This operation causes the target object to resume pulling events from the connected supplier. If the connection is not suspended the `ConnectionAlreadyActive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

D.13 Interface CosNotifyChannelAdmin::ProxyPushSupplier

interface **ProxyPushSupplier**
inherits from CosNotifyChannelAdmin::ProxySupplier, CosNotifyComm::PushSupplier

The `ProxyPushSupplier` interface manages connections from push consumers who wish to have unstructured events pushed on them by the channel.

Operations

connect_any_push_consumer

```
void connect_any_push_consumer(in CosEventComm::PushConsumer
push_consumer)
   raises(CosEventChannelAdmin::AlreadyConnected,
          CosEventChannelAdmin::TypeError);
```

Connects a consumer to the channel. If a consumer is already connected the `AlreadyConnected` exception is raised.

Parameters:

`push_consumer` - A reference to the consumer object. A nil reference is not permitted.

suspend_connection

```
void suspend_connection()
   raises(ConnectionAlreadyInactive,
          NotConnected);
```

This operation causes the target object to stop pushing events to the connected consumer. If the connection is already suspended the `ConnectionAlreadyInactive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

resume_connection

```
void resume_connection()
   raises(ConnectionAlreadyActive,
          NotConnected);
```

This operation causes the target object to resume pushing events to the connected consumer. If the connection is not suspended the `ConnectionAlreadyActive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

D.14 Interface CosNotifyChannelAdmin::StructuredProxyPushSupplier

interface **StructuredProxyPushSupplier**
inherits from CosNotifyChannelAdmin::ProxySupplier, CosNotifyComm::StructuredPushSupplier

The `StructuredProxyPushSupplier` interface manages connections from consumers who wish to have structured events pushed on them by the channel.

Operations

connect_structured_push_consumer

```
void connect_structured_push_consumer(in CosNotifyComm::Structured-
PushConsumer push_consumer)
   raises(CosEventChannelAdmin::AlreadyConnected,
          CosEventChannelAdmin::TypeError);
```

Connects a consumer to the channel. If a consumer is already connected the `AlreadyConnected` exception is raised.

Parameters:

`push_consumer` - A reference to the consumer object. A nil reference is not permitted.

suspend_connection

```
void suspend_connection()
   raises(ConnectionAlreadyInactive,
          NotConnected);
```

This operation causes the target object to stop pushing events to the connected consumer. If the connection is already suspended the `ConnectionAlreadyInactive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

resume_connection

```
void resume_connection()
   raises(ConnectionAlreadyActive,
          NotConnected);
```

This operation causes the target object to resume pushing events to the connected consumer. If the connection is not suspended the `ConnectionAlreadyActive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

D.15 Interface CosNotifyChannelAdmin::SequenceProxyPushSupplier

interface **SequenceProxyPushSupplier**
inherits from CosNotifyChannelAdmin::ProxySupplier, CosNotifyComm::SequencePushSupplier

The `SequenceProxyPushSupplier` interface manages connections from consumers who wish to have sequences of structured events pushed on them by the channel.

Operations

connect_sequence_push_consumer

```
void connect_sequence_push_consumer(in CosNotifyComm::SequencePushCon-
sumer push_consumer)
   raises(CosEventChannelAdmin::AlreadyConnected,
          CosEventChannelAdmin::TypeError);
```

Connects a consumer to the channel. If a consumer is already connected the `AlreadyConnected` exception is raised.

Parameters:

`push_consumer` - A reference to the consumer object. A nil reference is not permitted.

suspend_connection

```
void suspend_connection()
   raises(ConnectionAlreadyInactive,
          NotConnected);
```

This operation causes the target object to stop pushing events to the connected consumer. If the connection is already suspended the `ConnectionAlreadyInactive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

resume_connection

```
void resume_connection()
   raises(ConnectionAlreadyActive,
          NotConnected);
```

This operation causes the target object to resume pushing events to the connected consumer. If the connection is not suspended the `ConnectionAlreadyActive` exception is raised. If the target object is not connected to a supplier the `NotConnected` exception is raised.

D.16 Interface CosNotifyChannelAdmin::ConsumerAdmin

interface **ConsumerAdmin**
inherits from CosNotification::QoSAdmin, CosNotifyComm::NotifySubscribe, CosNotifyFilter::FilterAdmin, CosEventChannelAdmin::ConsumerAdmin

The `ConsumerAdmin` interface supports the creation of proxy suppliers.

Attributes

MyID
```
readonly attribute AdminID MyID;
```

The ID assigned to the target admin object by the channel.

MyChannel
```
readonly attribute EventChannel MyChannel;
```

A reference to the parent channel.

MyOperator
```
readonly attribute InterFilterGroupOperator MyOperator;
```

The `InterFilterGroupOperator` to be used when combining filter results from the target admin object and its child proxies.

priority_filter
```
attribute CosNotifyFilter::MappingFilter priority_filter;
```

Reference to an optional priority mapping filter.

lifetime_filter
```
attribute CosNotifyFilter::MappingFilter lifetime_filter;
```

Reference to an optional lifetime mapping filter.

pull_suppliers
```
readonly attribute ProxyIDSeq pull_suppliers;
```

A list of pull suppliers managed by the target admin object.

push_suppliers

```
readonly attribute ProxyIDSeq push_suppliers;
```

A list of push suppliers managed by the target admin object.

Operations

get_proxy_supplier

```
ProxySupplier get_proxy_supplier(in ProxyID proxy_id)
   raises(ProxyNotFound);
```

Obtains a reference to a proxy supplier with the given proxy ID.

Parameters:

`proxy_id` - The ID of the proxy to locate. A consumer admin object assigns an ID to each proxy it creates.

Returns:

If found, a reference to the proxy supplier is returned. Otherwise a `ProxyNotFound` exception is raised.

obtain_notification_pull_supplier

```
ProxySupplier obtain_notification_pull_supplier(in ClientType ctype,
                                                out ProxyID proxy_id)
   raises(AdminLimitExceeded);
```

Creates a new proxy pull supplier.

Parameters:

`ctype` - Specifies the client type. The returned proxy can be narrowed to a type suitable for the given client type.
`proxy_id` - Returns the ID assigned to the newly created proxy.

Returns:

A reference to a newly created proxy supplier is returned. This reference should be narrowed to the appropriate type before use. The `AdminLimitExceeded` exception is raised if creating a new proxy would exceed the limit for the target admin.

obtain_notification_push_supplier

```
ProxySupplier obtain_notification_push_supplier(in ClientType ctype,
                                                out ProxyID proxy_id)
```

```
raises(AdminLimitExceeded);
```

Creates a new proxy push supplier.

Parameters:

`ctype` - Specifies the client type. The returned proxy can be narrowed to a type suitable for the given client type.
`proxy_id` - Returns the ID assigned to the newly created proxy.

Returns:

A reference to a newly created proxy supplier is returned. This reference should be narrowed to the appropriate type before use. The `AdminLimitExceeded` exception is raised if creating a new proxy would exceed the limit for the target admin.

destroy

```
void destroy();
```

Destroys the target admin object and all proxies it is managing.

D.17 Interface CosNotifyChannelAdmin::SupplierAdmin

interface **SupplierAdmin**
inherits from CosNotification::QoSAdmin, CosNotifyComm::NotifyPublish, CosNotifyFilter::FilterAdmin, CosEventChannelAdmin::SupplierAdmin

The `SupplierAdmin` interface supports the creation of proxy consumers.

Attributes

MyID
```
readonly attribute AdminID MyID;
```

The ID assigned to the target admin object by the channel.

MyChannel
```
readonly attribute EventChannel MyChannel;
```

A reference to the parent channel.

MyOperator
```
readonly attribute InterFilterGroupOperator MyOperator;
```

The `InterFilterGroupOperator` to be used when combining filter results from the target admin object a its child proxies.

pull_consumers
```
readonly attribute ProxyIDSeq pull_consumers;
```

A list of pull consumers managed by the target admin object.

push_consumers
```
readonly attribute ProxyIDSeq push_consumers;
```

A list of push consumers managed by the target admin object.

Operations

get_proxy_consumer
```
ProxyConsumer get_proxy_consumer(in ProxyID proxy_id)
```

```
raises(ProxyNotFound);
```

Obtains a reference to a proxy consumer with the given proxy ID.

Parameters:

`proxy_id` - The ID of the proxy to locate. A supplier admin object assigns an ID to each proxy it creates.

Returns:

If found, a reference to the proxy consumer is returned. Otherwise a `ProxyNotFound` exception is raised.

obtain_notification_pull_consumer

```
ProxyConsumer obtain_notification_pull_consumer(in ClientType ctype,
                                                out ProxyID proxy_id)
   raises(AdminLimitExceeded);
```

Creates a new proxy pull consumer.

Parameters:

`ctype` - Specifies the client type. The returned proxy can be narrowed to a type suitable for the given client type.
`proxy_id` - Returns the ID assigned to the newly created proxy.

Returns:

A reference to a newly created proxy consumer is returned. This reference should be narrowed to the appropriate type before use. The `AdminLimitExceeded` exception is raised if creating a new proxy would exceed the limit for the target admin.

obtain_notification_push_consumer

```
ProxyConsumer obtain_notification_push_consumer(in ClientType ctype,
                                                out ProxyID proxy_id)
   raises(AdminLimitExceeded);
```

Creates a new proxy push consumer.

Parameters:

`ctype` - Specifies the client type. The returned proxy can be narrowed to a type suitable for the given client type.
`proxy_id` - Returns the ID assigned to the newly created proxy.

Returns:

A reference to a newly created proxy consumer is returned. This reference should be narrowed to the appropriate type before use. The `AdminLimitExceeded` exception is raised if creating a new proxy would exceed the limit for the target admin.

destroy

```
void destroy();
```

Destroys the target admin object and all proxies it is managing.

D.18 Interface CosNotifyChannelAdmin::EventChannel

interface **EventChannel**
inherits from CosNotification::QoSAdmin, CosNotification::AdminPropertiesAdmin, CosEventChannelAdmin::EventChannel

The `EventChannel` interface has operations which support the management of supplier and consumer admin objects.

Attributes

MyFactory

```
readonly attribute EventChannelFactory MyFactory;
```

A reference to the event channel factory which created the target object.

default_consumer_admin

```
readonly attribute ConsumerAdmin default_consumer_admin;
```

A reference to a default consumer admin which is created automatically when the channel is created.

default_supplier_admin

```
readonly attribute SupplierAdmin default_supplier_admin;
```

A reference to a default supplier admin which is created automatically when the channel is created.

default_filter_factory

```
readonly attribute CosNotifyFilter::FilterFactory
default_filter_factory;
```

A reference to the default filter factory.

Operations

new_for_consumers

```
ConsumerAdmin new_for_consumers(in InterFilterGroupOperator op,
                                out AdminID id);
```

Creates a new consumer admin.

Parameters:

`op` - The `InterFilterGroupOperator` to apply between filter results from the target object and subsequently created proxy objects.
`id` - The id assigned to the new consumer admin by the event channel.

Returns:

A reference to the newly created consumer admin is returned.

new_for_suppliers

```
SupplierAdmin new_for_suppliers(in InterFilterGroupOperator op,
                                out AdminID id);
```

Creates a new supplier admin.

Parameters:

`op` - The `InterFilterGroupOperator` to apply between filter results from the target object and subsequently created proxy objects.
`id` - The id assigned to the new supplier admin by the event channel.

Returns:

A reference to the newly created supplier admin is returned.

get_consumeradmin

```
ConsumerAdmin get_consumeradmin(in AdminID id)
   raises(AdminNotFound);
```

Obtains a reference to a consumer admin from an admin ID.

Parameters:

`id` - The ID of the admin for which a reference is required. The ID is originally assigned by the channel on creation of the admin.

Returns:

A reference to the consumer admin with the given ID. If no matching admin object is found an `AdminNotFound` exception is raised.

get_supplieradmin

```
SupplierAdmin get_supplieradmin(in AdminID id)
   raises(AdminNotFound);
```

Obtains a reference to a supplier admin from an admin ID.

Parameters:

`id` - The ID of the admin for which a reference is required. The ID is originally assigned by the channel on creation of the admin.

Returns:

A reference to the supplier admin with the given ID. If no matching admin object is found an `AdminNotFound` exception is raised.

get_all_consumeradmins

```
AdminIDSeq get_all_consumeradmins();
```

Obtains the IDs of all consumer admin objects associated with the target object.

Returns:

A sequence of admin IDs.

get_all_supplieradmins

```
AdminIDSeq get_all_supplieradmins();
```

Obtains the IDs of all supplier admin objects associated with the target object.

Returns:

A sequence of admin IDs.

D.19 Interface CosNotifyChannelAdmin::EventChannelFactory

interface **EventChannelFactory**

The `EventChannelFactory` interface contains operations which support the creation and management of Notification Service event channels.

Operations

create_channel

```
EventChannel create_channel(in CosNotification::QoSProperties
initial_qos,
                            in CosNotification::AdminProperties
initial_admin,
                            out ChannelID id)
   raises(CosNotification::UnsupportedQoS,
          CosNotification::UnsupportedAdmin);
```

Creates a new channel.

Parameters:

`initial_qos` - A sequence of QoS properties to be assigned to the new channel.
`initial_admin` - A sequence of administrative properties to be assigned to the new channel.
`id` - The ID assigned to the channel by the target object is returned in this parameter.

Returns:

A reference to the newly created channel is returned. If any of the QoS properties in `initial_QoS` are not supported an `UnsupportedQoS` exception is raised. If any of the administrative properties in `initial_admin` are not supported an `UnsupportedAdmin` exception is raised.

get_all_channels

```
ChannelIDSeq get_all_channels();
```

Obtains a list of all channels known to the factory.

Returns:

A sequence of IDs representing all channels currently managed by the target object.

get_event_channel

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

Obtains a channel reference from a channel ID.

Parameters:

`id` - The id of channel for which a reference is required.

Returns:

A reference to a channel with the corresponding ID. If no channel could be found with the given ID a `ChannelNotFound` exception is raised.

APPENDIX E CosNotifyComm Reference

E.1 Module CosNotifyComm

Exceptions

InvalidEventType

```
exception InvalidEventType
{
   CosNotification::EventType type;
};
```

Raised to indicate an event type name which contains syntax errors.

E.2 Interface CosNotifyComm::NotifyPublish

interface **NotifyPublish**

The `NotifyPublish` interface provides a method which suppliers can use to inform consumers of changes in the set of events offered.

Operations

offer_change

```
void offer_change(in CosNotification::EventTypeSeq added,
                  in CosNotification::EventTypeSeq removed)
   raises(InvalidEventType);
```

Reports changes in the event offering to consumers. If one or more of the event type names being added or removed is syntactically incorrect the `InvalidEventType` exception is raised.

Parameters:

`added` - A list of new event types being added to those currently offered.
`removed` - A list of event types no longer being supplied.

E.3 Interface CosNotifyComm::NotifySubscribe

interface **NotifySubscribe**

The `NotifySubscribe` interface provides a method which consumers can use to inform suppliers of the event types of interest.

Operations

subscription_change

```
void subscription_change(in CosNotification::EventTypeSeq added,
                         in CosNotification::EventTypeSeq removed)
   raises(InvalidEventType);
```

Reports changes in the event subscription to suppliers. If one or more of the event type names being added or removed is syntactically incorrect the `InvalidEventType` exception is raised.

Parameters:

`added` - A list of new event types being added to the current subscription.
`removed` - A list of event types being removed from the subscription.

E.4 Interface CosNotifyComm::PushConsumer

interface **PushConsumer**
inherits from CosNotifyComm::NotifyPublish, CosEventComm::PushConsumer

The `PushConsumer` interface is implemented and registered (connected) by clients who wish to have unstructured events pushed on them by the channel.

E.5 Interface CosNotifyComm::PullConsumer

interface **PullConsumer**
inherits from CosNotifyComm::NotifyPublish, CosEventComm::PullConsumer

The `PullConsumer` interface is implemented and registered (connected) by clients who wish to participate in subscription sharing and be notified when disconnected by the channel. Clients do not need to implement this interface to simply pull events.

E.6 Interface CosNotifyComm::PullSupplier

interface **PullSupplier**
inherits from CosNotifyComm::NotifySubscribe, CosEventComm::PullSupplier

The `PullSupplier` interface is implemented and registered (connected) by clients who wish to have unstructured events pulled from them by the channel.

E.7 Interface CosNotifyComm::PushSupplier

interface **PushSupplier**
inherits from CosNotifyComm::NotifySubscribe, CosEventComm::PushSupplier

The `PushSupplier` interface is implemented and registered (connected) by clients who wish to participate in subscription sharing and be notified when disconnected by the channel. Clients do not need to implement this interface to simply push events.

E.8 Interface CosNotifyComm::StructuredPushConsumer

interface **StructuredPushConsumer**
inherits from CosNotifyComm::NotifyPublish

The `StructuredPushConsumer` interface is implemented and registered (connected) by clients who wish to have structured events pushed on them by the channel.

Operations

push_structured_event

```
void push_structured_event(in CosNotification::StructuredEvent notifi-
cation)
   raises(CosEventComm::Disconnected);
```

Suppliers invoke this operation to pass structured event data to consumers. If communication is disconnected the `Disconnected` exception is raised.

Parameters:

`notification` - The structured event being pushed to the consumer.

disconnect_structured_push_consumer

```
void disconnect_structured_push_consumer();
```

Terminates communication between the target consumer and its supplier. Also frees resources allocated by the consumer.

E.9 Interface CosNotifyComm::StructuredPullConsumer

interface **StructuredPullConsumer**
inherits from CosNotifyComm::NotifyPublish

The `StructuredPullConsumer` interface is implemented and registered (connected) by clients who wish to participate in subscription sharing and be notified when disconnected by the channel. Clients do not need to implement this interface to simply pull events.

Operations

disconnect_structured_pull_consumer

```
void disconnect_structured_pull_consumer();
```

Terminates communication between the target consumer and its supplier. Also frees resources allocated by the consumer.

E.10 Interface CosNotifyComm::StructuredPullSupplier

interface **StructuredPullSupplier**
inherits from CosNotifyComm::NotifySubscribe

The `StructuredPullSupplier` interface is implemented and registered (connected) by clients who wish to have structured events pulled from them by the channel.

Operations

pull_structured_event

```
CosNotification::StructuredEvent pull_structured_event()
   raises(CosEventComm::Disconnected);
```

This method blocks the calling thread until the supplier has data available or an exception is raised.

Returns:
A structured event.

try_pull_structured_event

```
CosNotification::StructuredEvent try_pull_structured_event(out boolean
has_event)
   raises(CosEventComm::Disconnected);
```

This method does not block and can be used to poll a pull supplier for events.

Parameters:
`has_event` - Set to `TRUE` if there is an event available, `FALSE` otherwise.

Returns:
A structured event if `has_event` is `TRUE`, undefined otherwise.

disconnect_structured_pull_supplier

```
void disconnect_structured_pull_supplier();
```

Terminates communication between the target supplier and its consumer. Also frees resources allocated by the supplier.

E.11 Interface CosNotifyComm::StructuredPushSupplier

interface **StructuredPushSupplier**
inherits from CosNotifyComm::NotifySubscribe

The `StructuredPushSupplier` interface is implemented and registered (connected) by clients who wish to participate in subscription sharing and be notified when disconnected by the channel. Clients do not need to implement this interface to simply push events.

Operations

disconnect_structured_push_supplier

```
void disconnect_structured_push_supplier();
```

Terminates communication between the target supplier and its consumer. Also frees resources allocated by the supplier.

E.12 Interface CosNotifyComm::SequencePushConsumer

interface **SequencePushConsumer**
inherits from CosNotifyComm::NotifyPublish

The `SequencePushConsumer` interface is implemented and registered (connected) by clients who wish to have sequences of structured events pushed on them by the channel.

Operations

push_structured_events

```
void push_structured_events(in CosNotification::EventBatch notifica-
tions)
   raises(CosEventComm::Disconnected);
```

Suppliers invoke this operation to pass sequences of structured events to consumers. If communication is disconnected the `Disconnected` exception is raised.

Parameters:

`notifications` - The structured events being pushed to the consumer.

disconnect_sequence_push_consumer

```
void disconnect_sequence_push_consumer();
```

Terminates communication between the target consumer and its supplier. Also frees resources allocated by the consumer.

E.13 Interface CosNotifyComm::SequencePullConsumer

interface **SequencePullConsumer**
inherits from CosNotifyComm::NotifyPublish

The `SequencePullConsumer` interface is implemented and registered (connected) by clients who wish to participate in subscription sharing and be notified when disconnected by the channel. Clients do not need to implement this interface to simply pull events.

Operations

disconnect_sequence_pull_consumer

```
void disconnect_sequence_pull_consumer();
```

Terminates communication between the target consumer and its supplier. Also frees resources allocated by the consumer.

E.14 Interface CosNotifyComm::SequencePullSupplier

interface **SequencePullSupplier**
inherits from CosNotifyComm::NotifySubscribe

The `SequencePullSupplier` interface is implemented and registered (connected) by clients who wish to have sequences of structured events pulled from them by the channel.

Operations

pull_structured_events

```
CosNotification::EventBatch pull_structured_events(in long max_number)
   raises(CosEventComm::Disconnected);
```

This method blocks the calling thread until the supplier has data available or an exception is raised.

Parameters:

`max_number` - Indicates the maximum number of events to return.

Returns:

A sequence of structured events.

try_pull_structured_events

```
CosNotification::EventBatch try_pull_structured_events(in long
max_number,
                                                       out boolean
has_event)
   raises(CosEventComm::Disconnected);
```

This method does not block and can be used to poll a pull supplier for events.

Parameters:

`max_number` - Indicates the maximum number of events to return.
`has_event` - Set to `TRUE` if there is at lease one event is available, `FALSE` otherwise.

Returns:

A sequence of structured events if `has_event` is `TRUE`, undefined otherwise.

disconnect_sequence_pull_supplier

```
void disconnect_sequence_pull_supplier();
```

Terminates communication between the target supplier and its consumer. Also frees resources at the supplier.

E.15 Interface CosNotifyComm::SequencePushSupplier

interface **SequencePushSupplier**
inherits from CosNotifyComm::NotifySubscribe

The `SequencePushSupplier` interface is implemented and registered (connected) by clients who wish to participate in subscription sharing and be notified when disconnected by the channel. Clients do not need to implement this interface to simply push events.

Operations

disconnect_sequence_push_supplier

```
void disconnect_sequence_push_supplier();
```

Terminates communication between the target supplier and its consumer. Also frees resources allocated by the supplier.

APPENDIX F CosNotifyFilter Reference

F.1 Module CosNotifyFilter

This module provides interfaces which support all aspects of filter and mapping filter management.

Aliases

ConstraintID

```
typedef long ConstraintID;
```

Alias for a constraint ID.

ConstraintIDSeq

```
typedef sequence<ConstraintID> ConstraintIDSeq;
```

Alias for a sequence of constraint IDs.

ConstraintExpSeq

```
typedef sequence<ConstraintExp> ConstraintExpSeq;
```

Alias for a sequence of filter constraints.

ConstraintInfoSeq

```
typedef sequence<ConstraintInfo> ConstraintInfoSeq;
```

Alias for a sequence of constraint-ID pairs.

MappingConstraintPairSeq

```
typedef sequence<MappingConstraintPair> MappingConstraintPairSeq;
```

Alias for a sequence of mapping constraint pairs.

MappingConstraintInfoSeq

```
typedef sequence<MappingConstraintInfo> MappingConstraintInfoSeq;
```

Alias for a sequence of constraint-value pairs.

CallbackID

```
typedef long CallbackID;
```

Alias for a callback ID.

CallbackIDSeq

```
typedef sequence<CallbackID> CallbackIDSeq;
```

Alias for a sequence of callback IDs.

FilterID

```
typedef long FilterID;
```

Alias for a filter ID.

FilterIDSeq

```
typedef sequence<FilterID> FilterIDSeq;
```

Alias for a sequence of filter IDs.

Structs

ConstraintExp

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

A single filter constraint.

Members:

`event_types` - A sequence of event types which are matched against the event type information in the structured event header.
`constraint_expr` - A constraint expression which conforms to some constraint grammar.

ConstraintInfo

```
struct ConstraintInfo
{
   ConstraintExp constraint_expression;
   ConstraintID constraint_id;
};
```

Used to maintain an association between filter constraints and constraint IDs.

Members:

`constraint_expression` - A reference to the filter constraint.
`constraint_id` - The ID assigned to the filter constraint by the target object.

MappingConstraintPair

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

The mapping filter constraint-value pair.

Members:

`constraint_expression` - A filter constraint.
`result_to_set` - The result to return from a match operation which matches on the corresponding constraint.

MappingConstraintInfo

```
struct MappingConstraintInfo
{
   ConstraintExp constraint_expression;
   ConstraintID constraint_id;
   any value;
};
```

Used to maintain an association between mapping filter constraints and constraint IDs.

Members:

`constraint_expression` - A filter constraint.

`constraint_id` - A unique ID assigned to the constraint-value pair by the target mapping filter object.

`value` - The result to return from a match operation which matches on the corresponding constraint.

Exceptions

UnsupportedFilterableData

```
exception UnsupportedFilterableData
{
};
```

Raised during a match operation if the input event contains data that the match operation is not designed to handle.

InvalidGrammar

```
exception InvalidGrammar
{
};
```

Raised during filter creation if an invalid constraint grammar is specified.

InvalidConstraint

```
exception InvalidConstraint
{
   ConstraintExp constr;
};
```

Raised during the addition or modification of constraints if the new constraint does not conform to the specified grammar for the target filter object.

DuplicateConstraintID

```
exception DuplicateConstraintID
{
   ConstraintID id;
};
```

Not used.

ConstraintNotFound

```
exception ConstraintNotFound
```

```
{
   ConstraintID id;
};
```

Raised when an operation cannot find a constraint with the given ID.

CallbackNotFound

```
exception CallbackNotFound
{
};
```

Raised when an operation cannot find a callback with the given ID.

InvalidValue

```
exception InvalidValue
{
   ConstraintExp constr;
   any value;
};
```

Raised if the datatype of a value in an input constraint-value pair does not match the `value_type` for the target mapping filter object.

FilterNotFound

```
exception FilterNotFound
{
};
```

Indicates that a reference for a specified filter was not found.

F.2 Interface CosNotifyFilter::Filter

interface **Filter**

The `Filter` interface manages groups of filter constraint expressions and has operations which evaluate events against these constraints.

Attributes

constraint_grammar

```
readonly attribute string constraint_grammar;
```

The constraint grammar specified during creation of the filter. All constraints for the target filter object must be expressed in this grammar.

Operations

add_constraints

```
ConstraintInfoSeq add_constraints(in ConstraintExpSeq constraint_list)
   raises(InvalidConstraint);
```

Add a list of filter constraints to the target filter object. This operation is incremental in that new constraints are appended to the existing list of constraints.

Parameters:

`constraint_list` - The list of constraints to be added to the target filter object.

Returns:

The target filter object assigns an ID to each constraint. This list of constraint-ID pairs is returned. If any of the constraints violate the constraint grammar an `InvalidConstraint` exception is raised.

modify_constraints

```
void modify_constraints(in ConstraintIDSeq del_list,
                        in ConstraintInfoSeq modify_list)
   raises(InvalidConstraint,
          ConstraintNotFound);
```

Modifies the list of constraints associated with the target filter object. If one or more of the IDs in either of the two lists are not found the `ConstraintNotFound` exception is raised.

Parameters:

`del_list` - A list of constraint IDs representing constraints to remove from the target filter object.

`modify_list` - A list of constraint IDs and constraint expressions. Constraints which exist in the target filter object are modified to those in the list with the same constraint ID. If a constraint in this list does not conform to the constraint grammar for the target filter object, an `InvalidConstraint` exception is raised.

get_constraints

```
ConstraintInfoSeq get_constraints(in ConstraintIDSeq id_list)
   raises(ConstraintNotFound);
```

Retrieves a set of constraints from the target filter object.

Parameters:

`id_list` - A list of constraint IDs representing the constraints to be retrieved.

Returns:

The constraints associated with the target filter object with the given IDs. If one or more of the IDs are not found the `ConstraintNotFound` exception is raised.

get_all_constraints

```
ConstraintInfoSeq get_all_constraints();
```

Retrieve all constraints associated with the target filter object.

Returns:

All constraints associated with the target filter object.

remove_all_constraints

```
void remove_all_constraints();
```

Remove all constraints associated with the target filter object.

destroy

```
void destroy();
```

Destroys the target filter object.

match

```
boolean match(in any filterable_data)
   raises(UnsupportedFilterableData);
```

Compare the filter constraints from the target filter object with the supplied event.

Parameters:

`filterable_data` - The event to be evaluated in the form of a `CORBA::Any`.

Returns:

Returns `TRUE` if the event satistifes at least one constraint, `FALSE` otherwise. If the filterable data of the input event contains data that the match operation cannot handle, an `UnsupportedFilterableData` exception is raised.

match_structured

```
boolean match_structured(in CosNotification::StructuredEvent
filterable_data)
   raises(UnsupportedFilterableData);
```

Compare the filter constraints from the target filter object with the supplied event.

Parameters:

`filterable_data` - The event to be evaluated in the form of a structured event.

Returns:

Returns `TRUE` if the event satistifes at least one constraint, `FALSE` otherwise. If the filterable data of the input event contains data that the match operation cannot handle an `UnsupportedFilterableData` exception is raised.

match_typed

```
boolean match_typed(in CosNotification::PropertySeq filterable_data)
   raises(UnsupportedFilterableData);
```

Not implemented.

attach_callback

```
CallbackID attach_callback(in CosNotifyComm::NotifySubscribe callback);
```

Allows objects supporting the `NotifySubscribe` interface (proxy suppliers and consumer admins) to register with the target filter object. Registered objects are notified when the set of event types required by the filter constraints changes.

Parameters:

`callback` - A reference to an object interested in subscription changes.

Returns:
The target filter object assigns and returns a unique ID to each registered callback.

detach_callback

```
void detach_callback(in CallbackID callback)
   raises(CallbackNotFound);
```

Removes a callback previously registered with `attach_callback`.

Parameters:
`callback` - The ID of the callback to be removed. The `CallbackNotFound` exception is raised of the target object does not contain a reference with the given ID.

get_callbacks

```
CallbackIDSeq get_callbacks();
```

Retrieve a list of all callbacks registered with the target filter object.

Returns:
A list of IDs representing all callbacks currently registered.

F.3 Interface CosNotifyFilter::MappingFilter

interface **MappingFilter**

The `MappingFilter` interface manages groups of mapping filter constraint-value pairs and has operations which evaluate events against these constraints.

Attributes

constraint_grammar

```
readonly attribute string constraint_grammar;
```

The constraint grammar specified during creation of the filter. All constraints for a filter object must be expressed in this grammar.

value_type

```
readonly attribute TypeCode value_type;
```

Identifies the datatype of the property value which the mapping filter affects.

default_value

```
readonly attribute any default_value;
```

This parameter is returned as the result of a match operation for which the given event satisfied none of the constraints associated with the target mapping filter object.

Operations

add_mapping_constraints

```
MappingConstraintInfoSeq add_mapping_constraints(in MappingConstraint-
PairSeq pair_list)
   raises(InvalidConstraint,
          InvalidValue);
```

Add a list of mapping filter constraints to the target mapping filter object. This operation is incremental in that new constraints are appended to the existing list of constraints.

Parameters:

`pair_list` - The list of constraint-value pairs to be added to the target filter object.

Returns:

The target filter object assigns an ID to each constraint-value pair. The input list is returned along with the ID assigned to each constraint-value pair. If any of the constraints violate the constraint grammar an `InvalidConstraint` exception is raised. If any of the values in the list of constraint-value pairs are not of the same type as the value_type for the target filter object, an `InvalidValue` exception is raised.

modify_mapping_constraints

```
void modify_mapping_constraints(in ConstraintIDSeq del_list,
                                in MappingConstraintInfoSeq modify_list)
   raises(InvalidConstraint,
          InvalidValue,
          ConstraintNotFound);
```

Modifies the list of constraint-value pairs associated with the target filter object. If one or more of the IDs in either of the two lists are not found the `ConstraintNotFound` exception is raised.

Parameters:

`del_list` - A list of constraint IDs representing constraint-value pairs to remove from the target filter object.

`modify_list` - A list of constraint IDs and constraint-value pairs. Constraints which exist in the target filter object are modified to those in the list with the same constraint ID. Both the constraint and value types may be modified. If a constraint in this list does not conform to the constraint grammar for the target filter object, an `InvalidConstraint` exception is raised. Likewise if a value in this list is not of the same type as the `value_type` for the target filter object, an `InvalidValue` exception is raised.

get_mapping_constraints

```
MappingConstraintInfoSeq get_mapping_constraints(in ConstraintIDSeq
id_list)
   raises(ConstraintNotFound);
```

Retrieves a set of constraint-value pairs from the target filter object.

Parameters:

`id_list` - A list of constraint IDs representing the constraint-value pairs to be retrieved.

Returns:

The constraint-value pairs associated with the target filter object with the given IDs. If one or more of the IDs are not found the `ConstraintNotFound` exception is raised.

get_all_mapping_constraints

```
MappingConstraintInfoSeq get_all_mapping_constraints();
```

Retrieve all constraint-value pairs associated with the target filter object.

Returns:
All constraint-value pairs associated with the target filter object.

remove_all_mapping_constraints

```
void remove_all_mapping_constraints();
```

Remove all constraint-value pairs associated with the target filter object.

destroy

```
void destroy();
```

Destroys the target filter object.

match

```
boolean match(in any filterable_data,
              out any result_to_set)
   raises(UnsupportedFilterableData);
```

Compare the filter constraints from the target filter object with the supplied event.

Parameters:
`filterable_data` - The event to be evaluated in the form of a `CORBA::Any`.
`result_to_set` - If the match is successful, that is the return result is `TRUE`, this parameter is set to the value paired with the matching constraint. Otherwise if the match fails, that is the return result is `FALSE`, this parameter is set to the `default_value` for the target filter object.

Returns:
Returns `TRUE` if the event satistifes at least one constraint, `FALSE` otherwise. If the filterable data of the input event contains data that the match operation cannot handle, an `UnsupportedFilterableData` exception is raised.

match_structured

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

Compare the filter constraints from the target filter object with the supplied event.

Parameters:

`filterable_data` - The event to be evaluated in the form of a structured event.
`result_to_set` - If the match is successful, that is the return result is `TRUE`, this parameter is set to the value paired with the matching constraint. Otherwise if the match fails, that is the return result is `FALSE`, this parameter is set to the `default_value` for the target filter object.

Returns:

Returns `TRUE` if the event satistifes at least one constraint, `FALSE` otherwise. If the filterable data of the input event contains data that the match operation cannot handle, an `UnsupportedFilterableData` exception is raised.

match_typed

```
boolean match_typed(in CosNotification::PropertySeq filterable_data,
                    out any result_to_set)
raises(UnsupportedFilterableData);
```

Not Implemented.

F.4 Interface CosNotifyFilter::FilterFactory

interface **FilterFactory**

The `FilterFactory` interface includes operations which support the creation of filter objects and mapping filter objects.

Operations

create_filter

```
Filter create_filter(in string constraint_grammar)
   raises(InvalidGrammar);
```

Creates a new filter object.

Parameters:

`constraint_grammar` - The constraint grammar to be used for constraint expressions.

Returns:

A new filter object is returned. If an unknown constraint grammar is specified an `InvalidGrammar` exception is raised.

create_mapping_filter

```
MappingFilter create_mapping_filter(in string constraint_grammar,
                                    in any default_value)
   raises(InvalidGrammar);
```

Creates a new mapping filter object.

Parameters:

`constraint_grammar` - The constraint grammar to be used for constraint expressions.
`default_value` - The default value returned by a match operation on the target mapping filter.

Returns:

A new filter object is returned. If an unknown constraint grammar is specified an `InvalidGrammar` exception is raised.

F.5 Interface CosNotifyFilter::FilterAdmin

interface **FilterAdmin**

The `FilterAdmin` interface supports the management of filter objects.

Operations

add_filter

```
FilterID add_filter(in Filter new_filter);
```

Adds a filter to the target object.

Parameters:

`new_filter` - The filter object to be added to the target object.

Returns:

The ID assigned to the filter by the target object is returned.

remove_filter

```
void remove_filter(in FilterID filter)
   raises(FilterNotFound);
```

Remove a filter from the target object, the filter itself is not destroyed. If the specified filter is not found a `FilterNotFound` exception is raised.

Parameters:

`filter` - The ID of the filter to remove.

get_filter

```
Filter get_filter(in FilterID filter)
   raises(FilterNotFound);
```

Retrieves a reference for the filter with the given filter ID from the target object.

Parameters:

`filter` - The ID of the filter to locate.

Returns:

A reference to a filter object is returned. If a filter with a given ID could not a be found a `FilterNotFound` exception is raised.

get_all_filters

```
FilterIDSeq get_all_filters();
```

Retrieve a list of all filters associated with the target object.

Returns:

A list of filter IDs is returned.

remove_all_filters

```
void remove_all_filters();
```

Remove all filters associated with the target object.

APPENDIX G OBNotify Reference

G.1 Module OBNotify

Proprietary ORBacus Notify QoS settings.

Constants

PullInterval

```
const string PullInterval = "PullInterval";
```

The amount of time the service pauses between pull requests. The value of this property is of type `TimeBase::TimeT`, with a default of 1 second.

RetryTimeout

```
const string RetryTimeout = "RetryTimeout";
```

Specifies the initial amount of time as a `TimeBase::TimeT` that the service will wait before retrying a failed client communications attempt. The default value is 1 second.

RetryMultiplier

```
const string RetryMultiplier = "RetryMultiplier";
```

After each consecutive expiration of the retry timeout, the timeout value will be multiplied by

this factor. This value is a double and has a valid range of 1.0 to 2.0 inclusive. The default value is 1.0.

MaxRetries

```
const string MaxRetries = "MaxRetries";
```

The maximum number of retries that will be performed before the proxy ceases making requests to the connected consumer or supplier. The proxy then disconnects and destroys itself. The default value is 0, which means unlimited retry.

MaxRetryTimeout

```
const string MaxRetryTimeout = "MaxRetryTimeout";
```

The upper limit, as a `TimeBase::TimeT`, for increasing the retry interval. After this duration has been reached the retry interval will stay constant until success or until `OBNotify::MaxRetries` has been reached. The default value is 60 seconds.

RequestTimeout

```
const string RequestTimeout = "RequestTimeout";
```

The amount of time (`TimeBase::TimeT`) permitted for a blocking request on a client to return before a timeout. The default value is 5 seconds.

References

[1] Object Management Group. 2000. *Notification Service Specification.* ftp://ftp.omg.org/pub/docs/formal/00-06-20.pdf. Framingham, MA: Object Management Group.

[2] Object Management Group. 2001. *Event Service Specification.* ftp://ftp.omg.org/pub/docs/formal/01-03-01.pdf. Framingham, MA: Object Management Group.

[3] Object Management Group. 1999. *The Common Object Request Broker: Architecture and Specification*. Revision 2.3.1. ftp://ftp.omg.org/pub/docs/formal/99-10-07.pdf. Framingham, MA: Object Management Group.

[4] IONA Technologies, Inc. 2001. *ORBACUS User's Manual*. Waltham, MA: IONA Technologies, Inc.