cosNotification

cosNotification 1.1.13
May 17 2010

May 17 2010

1 User's Guide

The *cosNotification* application is an Erlang implementation of the OMG CORBA Notification Service.

1.1 The cosNotification Application

1.1.1 Content Overview

The cosNotification documentation is divided into three sections:

- PART ONE - The User's Guide
  Description of the cosNotification Application including services and a small tutorial demonstrating the development of a simple service.
- PART TWO - Release Notes
  A concise history of cosNotification.
- PART THREE - The Reference Manual
  A quick reference guide, including a brief description, to all the functions available in cosNotification.

1.1.2 Brief Description of the User's Guide

The User's Guide contains the following parts:

- cosNotification overview
- cosNotification installation
- A tutorial example

1.2 Introduction to cosNotification

1.2.1 Overview

The cosNotification application is a Notification Service compliant with the **OMG** Notification Service CosNotification.

Purpose and Dependencies

cosNotification is dependent on *Orber-3.1.7* or later, which provides CORBA functionality in an Erlang environment, *cosTime-1.0.1* or later and IDL-files to be compiled using *IC-4.0.4* or later.

Prerequisites

To fully understand the concepts presented in the documentation, it is recommended that the user is familiar with distributed programming, CORBA and the Orber application.

Recommended reading includes books recommended by the *OMG* and *Open Telecom Platform Documentation Set*. It is also helpful to have read *Concurrent Programming in Erlang*.

1.3 Installing cosNotification

1.3.1 Installation Process

This chapter describes how to install *cosNotificationApp* in an Erlang Environment.

Preparation

Before starting the installation process for cosNotification, the application Orber must be running.

Configuration

When using the Notification Service the `cosNotification` application first must be installed using `cosNotificationApp:install()` or `cosNotificationApp:install(Seconds)`, followed by `cosNotificationApp:start()`.

Then the *Event Channel Factory* must be started:

- `cosNotificationApp:start_global_factory()` - starts and returns a reference to a factory using default configuration parameters. This operation should be used for a multi-node Orber.
- `cosNotificationApp:start_global_factory(Options)` - starts and returns a reference to a factory using given configuration parameters. This operation should be used for a multi-node Orber.
- `cosNotificationApp:start_factory()` - starts and returns a reference to a factory using default configuration parameters.
- `cosNotificationApp:start_factory(Options)` - starts and returns a reference to a factory using given configuration parameters.

The following options exist:

- `{pullInterval, Seconds}` - determine how often Proxy Pull Consumers will check for new events with the client application. The default value is 20 seconds.
- `{filterOp, OperationType}` - determine which type of Administrator objects should be started, i.e., `'OR_OP'` or `'AND_OP'`. The default value is `'OR_OP'`.
- `{timeService, TimeServiceObj | 'undefined'}` - to be able to use Start and/or Stop QoS this option must be used. See the function `start_time_service/2` in the `cosTime` application. The default value is `'undefined'`.
- `{filterOp, OperationType}` - determine which type of Administrator objects should be started, i.e., `'OR_OP'` or `'AND_OP'`. The default value is `'OR_OP'`.
- `{gcTime, Seconds}` - this option determines how often, for example, proxies will garbage collect expired events. The default value is 60.
- `{gcLimit, Amount}` - determines how many events will be stored before, for example, proxies will garbage collect expired events. The default value is 50. This option is tightly coupled with the QoS property `MaxEventsPerConsumer`, i.e., the `gcLimit` should be less than `MaxEventsPerConsumer` and greater than 0.

It is possible to define a set of global configuration parameters:

Key	*Range*	*Default*
type_check	true | false	true
notify	atom() | false	false
max_events	integer() > 0	50

interval_events	integer() > 0	10000 milliseconds
timeout_events	integer() > interval_events	3000000 milliseconds

Table 3.1: Global Configuration Parameters

Comments on the table 'Global Configuration Parameters':

type_check
Determine if supplied IOR:s shall be type checked, i.e. invoking corba_object:is_a/2, or not.

notify
The given value shall point to an existing module exporting a function (arity 1) called *terminated*. This operation is invoked when a proxy terminates and the argument is a list containing `{proxy, IOR}`, `{client, IOR}` and `{reason, term()}`. The return value is ignored.

max_events
If a supplier proxy has not been able to push events to a consumer and the queue exceeds this limit, then the proxy will terminate. For this option to have any effect, the `EventReliability` and `ConnectionReliability` QoS parameters must be set to `Persistent`. For more information, see also the *QoS* chapter.

interval_events
The same requirements as for `max_events`. When a supplier proxy detects problems when trying to push events, this parameter determines how often it should try to call the consumer.

timeout_events
The same requirements as for `max_events`. If the proxy has not been able to contact the consumer and this time-limit is reached, then the proxy will terminate.

The Factory is now ready to use. For a more detailed description see *Examples*.

1.4 The Notification Service Components

1.4.1 The Notification Service Components

This chapter describes the Notification Service Components and how they interact.

Components

There are seven components in the OMG Notification Service architecture. These are described below:

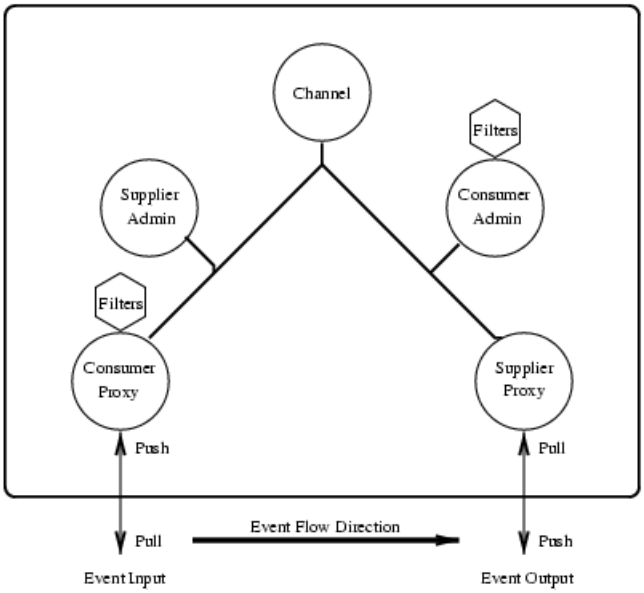

Figure 4.1: Figure 1: The Notification Service Components.

- *Event Channel:* acts as a factory for Administrator objects. Allows clients to set Administrative Properties.
- *Supplier Administrators:* acts as a factory for Proxy Consumers. Administrators are started as `'AND_OP'-` or `'OR_OP'-`type, which determines if events must be validated using both the Administrators associated Filter and/or its Proxy children Filters.
- *Consumer Administrators:* acts in the same way as Supplier Administrators but handle Proxy Suppliers.
- *Consumer Proxy:* is connected to a client application. Can be started as `Pull` or `Push` object. If the proxy is Push style the client application must push events to the Proxy, otherwise the Proxy is supposed to Pull events. The `CosNotification::AdminProperties` is used to set the pacing interval.
- *Supplier Proxy:* Acts in a similar way as the Consumer Proxy, but if started as a `Push` proxy it will push events to the client application.
- *Filters:* used to filter events. May be associated with Proxies and Administrators.
- *Mapping Filters:* used to override events Quality of Service settings. Can only be associated with Consumer Administrators and Proxy Suppliers.

When a Proxy is started it is set to accept `CORBA::Any`, `CosNotification::StructuredEvent` or `CosNotification::EventBatch` (a sequence of structured events).

If a Proxy is supposed to deliver structured events to a client application and receives an `CORBA::Any` event, the event is converted to a structured event with `type_name` set to `"%ANY"` and the event is stored in `remainder_of_body`.

If a Proxy is supposed to deliver `CORBA::Any` events to a client application and receives a structured event, the event is stored in an Any type. The Any Type Code will be equal to the `CosNotification::StructuredEvent` Type Code.

1.5 Filters and the Constraint Language BNF

1.5.1 Filters and the Constraint Language BNF

This chapter describes, the grammar supported by *CosNotifyFilter_Filter* and *CosNotifyFilter_MappingFilter*, and how to create and use filter objects.

How to create filter objects

To be able to filter events we must create a filter and associate it with one, or more, of the administrative or proxy objects. In the example below, we choose to associate the filter with a ConsumerAdmin object.

```
FilterFactory = cosNotificationApp:start_filter_factory(),
Filter = 'CosNotifyFilter_FilterFactory':
                    create_filter(FilterFactory,"EXTENDED_TCL"),
ConstraintInfoSeq = 'CosNotifyFilter_Filter':
                    add_constraints(Filter, ConstraintExpSeq),
FilterID = 'CosNotifyChannelAdmin_ConsumerAdmin':
                    add_filter(AdminConsumer, Filter),
```

`"EXTENDED_TCL"` is the only grammar supported by Orber Notification Service.

Depending on which operation type the Admin object uses, i.e., `'AND_OP'` or `'OR_OP'`, events will be tested using the associated filter. The operation properties are:

- 'AND_OP' - must be approved by the proxy's *and* its parent admin's filters. If all filters associated with an object (Admin or Proxy) return false the event will be discarded. In this situation it is pointless to try and verify with the other object's associated filters since the outcome still would be the same.
- 'OR_OP' - if one of the object's (Admin or Proxy) filters return true, the event will not be checked against any other filter associated with a proxy or its parent admin. If a object's associated filters all return false, the event will be forwarded to related proxies/admins, and tested against any associated filters.

Initially, filters are empty and will always return true. Hence, we must add constraints by using `'CosNotifyFilter_Filter':add_constraints/2`. As input, the second argument must be a sequence of:

```
#'CosNotifyFilter_ConstraintExp'{
       event_types = [#'CosNotification_EventType'{
                              domain_name = string(),
                              type_name = string()}],
       constraint_expr = string()}
```

The `event_types` describes which types of events that should be matched using the associated `constraint_expr`.

If a constraint expression is supposed to apply for all events, then the `type_name` can be set to the special event type `%ALL` in a constraint's event type sequence. The `domain_name` should be `""` or `"*"`.

In the following sections we will take a closer look on how to write constraint expressions.

The CosNotification Constraint Language

The constraint language supported by the Notification Service is:

```
<constraint> := /* empty */
    | <bool>

<bool> := <bool_or>

<bool_or> := <bool_or> or <bool_and>
    | <bool_and>
```

```
<bool_and> := <bool_and> and <bool_compare>
    | <bool_compare>

<bool_compare> := <expr_in> == <expr_in>
    | <expr_in> != <expr_in>
    | <expr_in> < <expr_in>
    | <expr_in> <= <expr_in>
    | <expr_in> > <expr_in>
    | <expr_in> >= <expr_in>
    | <expr_in>

<expr_in> := <expr_twiddle> in <Ident> /* sequence only */
    | <expr_twiddle>
    | <expr_twiddle> in $ <Component> /* sequence only */

<expr_twiddle> := <expr> ~ <expr> /* string data types only */
    | <expr>

<expr> := <expr> + <term>
    | <expr> - <term>
    | <term>

<term> := <term> * <factor_not>
    | <term> / <factor_not>
    | <factor_not>

<factor_not> := not <factor>
    | <factor>

<factor> := ( <bool_or> )
    | exist <Ident>
    | <Ident>
    | <Number>
    | - <Number>
    | <String>
    | TRUE
    | FALSE
    | + <Number>
    | exist $ <Component>
    | $ <Component>
    | default $ <Component> /* discriminated unions only */

<Component> := /* empty */
    | . <CompDot>
    | <CompArray>
    | <CompAssoc>
    | <Ident> <CompExt> /* run-time variable */

<CompExt> := /* empty */
    | . <CompDot>
    | <CompArray>
    | <CompAssoc>

<CompDot> := <Ident> <CompExt>
    | <CompPos>
    | <UnionPos>
    | _length   /* only valid for arrays or sequences */
    | _d        /* discriminated unions only */
    | _type_id  /* only valid if possible to obtain */
    | _repos_id /* only valid if possible to obtain */

<CompArray> := [ <Digits> ] <CompExt>

<CompAssoc> := ( <Ident> ) <CompExt>
```

```
<CompPos> := <Digits> <CompExt>

<UnionPos> := ( <UnionVal> ) <CompExt>

<UnionVal> := /* empty */
    | <Digits>
    | - <Digits>
    | + <Digits>
    | <String>

/* Character set issues */
<Ident> :=<Leader> <FollowSeq>
    | \ < Leader> <FollowSeq>

<FollowSeq> := /* <empty> */
    | <FollowSeq> <Follow>

<Number> := <Mantissa>
    | <Mantissa> <Exponent>

<Mantissa> := <Digits>
    | <Digits> .
    | . <Digits>
    | <Digits> . <Digits>

<Exponent> := <Exp> <Sign> <Digits>

<Sign> := +
    | -

<Exp> := E
    | e

<Digits> := <Digits> <Digit>
    | <Digit>

<String> := ' <TextChars> '

<TextChars> := /* <empty> */
    | <TextChars> <TextChar>

<TextChar> := <Alpha>
    | <Digit>
    | <Other>
    | <Special>

<Special> := \\
    | \'

<Leader> := <Alpha>

<Follow> := <Alpha>
    | <Digit>
    | _

<Alpha> is the set of alphabetic characters [A-Za-z]
<Digit> is the set of digits [0-9]
<Other> is the set of ASCII characters that are not <Alpha>, <Digit>, or <Special>
```

In the absence of parentheses, the following precedence relations hold :

- `()`, `exist`, `default`, `unary-sign`
- `not`

- `*, /`
- `+, -`
- `~`
- `in`
- `==, !=, <, <=, >, >=`
- `and`
- `or`

The Constraint Language Data Types

The Notification Service Constraint Language, defines how to write constraint expressions, which can be used to filter events. The representation does, however, differ slightly from ordinary Erlang terms.

When creating a `ConstraintExp`, the field `constraint_expr` must be set to contain a string, e.g., `"1 < 2"`. The Notification Service Constraint Language, is designed to be able to filter structured and unstructured events using the same constraint expression. The Constraint Language Types and Operations can be divided into two sub-groups:

- Basic - arithmetics, strings, constants, numbers etc.
- Complex - accessing members of complex data types, such as unions.

Some of the basic types, e.g., integer, are self explanatory. Hence, they are not described further.

Type/Operation	*Examples*	*Description*
`string`	`"'MyString'"`	Strings are represented as a sequence of zero or more `<TextChar>`s enclosed in single quotes, e.g., `'string'`.
`~`	`"'Sring1' ~ 'String2'"`	The operator ~is called the substring operator and mean "String1 is contained within String2".
`boolean`	`"TRUE == (('lang' ~ 'Erlang' + 'fun' ~ 'functional') >= 2)"`	Booleans may only be TRUE or FALSE, i.e., only capital letters. Expressions which evaluate to TRUE or FALSE can be summed up and matched, where TRUE equals 1 and FALSE 0.
`sequence`	`"myIntegerSequence[2]"`	The BNF use C/C++ notation, i.e., the example will return the *third*element.
`_length`	`"myIntegerSequence._length"`	Returns the length of an sequence or array.
`in`	`"'Erlang' in $.FunctionalLanguages-StringSeq"`	Returns `TRUE`if a given element is found in the given sequence. The element must be of a simple type and the same as the sequence is defined to contain.

`$`	`"$ == 40"`	Denote the current event as well as any run-time variables. If the event is unstructured and its contained value 40, the example will return `TRUE`.
`.`	`"$.MyStructMember == 40"`	The structure member operator `.` may be used to reference its members when the data refers to a named structure, discriminated union, or CORBA::Any data structure.
`_type_id`	`"$._type_id == 'MyStruct'"`	Returns the unscoped IDL type name of the component. This operation is only valid if said information can be obtained.
`_repos_id`	`"$._repos_id == 'IDL:MyModule/MyStruct:1.0'"`	Returns the RepositoryId of the component. This operation is only valid if said information can be obtained.
`_d`	`"$.eventUnion._d"`	May only be used when accessing discriminated unions and refers to the discriminator.
`exist`	`"exist $.eventUnion._d and $.eventUnion._d == 10"`	To avoid that a filtering of an event fails due to that, for example, we try to compare a union discriminator which does not exist, we can use this operator.
`default`	`"default $.eventUnion._d"`	If the `_d` operation is in conjunction with the `default` operation, TRUE will be returned if the union has a default member that is active.
`union`	`"$.(0) == 5"` eq. `"$.('zero') == 5"`	When the component refers to a union, with one of the cases defined as `case 0: short zero;`, we use `0` or `'zero'`. The result of the example is `TRUE` if the union has a discriminator set to `0` and the value `5`. If more than one case is defined to be `'zero'`, `$.('zero')` accepts both; `$.(0)` only returns `TRUE` if the discriminator is set to `0`. Leaving out the identifier, i.e., `$.()`, refers to the default value.

`name-value pairs`	`"$.NameValueSeq('myID') == 5"` eq. `"$.NameValueSeq[1].name == 'myID' and $.NameValueSeq[1].value == 5"`	The Notification service makes extensive use of `name-value pairs` sequences within structured events, which allow us to via the identifier `name` access its `value`, as shown in the example.

Table 5.1: Table 1: Type and Operator Examples

In the next section we will take a closer look at how it is possible to write constraints using different types of notation etc.

Accessing Data In Events

To filter events, the supplied constraints must describe the contents of the events and desired values. We can, for example, state that we are only interested in receiving events which are of type *CommunicationsAlarm*. To be able to achieve this, the constraint must contain information that points out which fields to compare with. Figure one illustrates a conceptual overview of a structured event. The exact definition is found in the `CosNotification.idl` file.

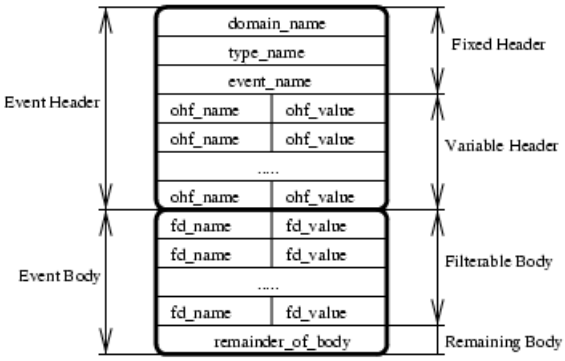

Figure 5.1: Figure 1: The structure of a structured event.

The Notification Service supports different constraint expressions notation:

- Fully scoped, e.g., "$.header.fixed_header.event_type.type_name == 'CommunicationsAlarm'"
- Short hand, e.g., "$type_name == 'CommunicationsAlarm'"
- Positional Notation, e.g., "$.0.0.0.1 == 'CommunicationsAlarm'"

Note:

Which notation to use is up to the user, however, the fully scoped may be easier to understand, but in some cases, if received from an ORB that do not populate ID:s of named parts, the positional notation is the only option.

Note:

If a constraint, which access fields in a structured event structure, is supposed to handle unstructured events as well, the CORBA::Any must contain the same type of members.

How to filter against the fixed header fields, is described in the table below.

Field	Fully Scoped Constraint	Short Hand Constraint
type_name	"$.header.fixed_header.event_-type.type_name == 'Type'"	"$type_name == 'Type'"
domain_name	"$.header.fixed_header.event_-type.domain_name == 'Domain'"	"$domain_name == 'Domain'"
event_name	"$.header.fixed_header.event_name == 'Event'"	"$event_name == 'Event'"

Table 5.2: Table 2: Fixed Header Constraint Examples

If we are only interested in receiving events regarding 'Domain', 'Event' and 'Type', the constraint can look like `"$domain_name == 'Domain' and $event_name == 'Event' and $type_name == 'Type'"`.

The variable event header consists of a sequence of *name-value pairs*. One way to filter on these are to use a constraint that looks like `"($.header.variable_header[1].name == 'priority' and $.header.variable_header[1].value > 0)"`. An easier way to accomplish the same result is to use a constraint that treats the name-value pair as an associative array, i.e., when given a name the corresponding value is returned. Hence, instead we can use `"$.header.variable_header(priority) > 0"`.

Accessing the event body is done in the same way as for the event header fields. The user must, however, be aware of, that if a run-time variable (`$variable`) is used data in the event header may take precedence. The order of precedence is:

- Reserved, e.g., `$curtime`
- A simple-typed member of `$.header.fixed_header`.
- Properties in `$.header.variable_header`.
- Properties in `$.filterable_data`.
- If no match is found it is translated to `$.variable`.

Mapping Filters

Mapping Filters may only be associated with Consumer Administrators or Proxy Suppliers. The purpose of a Mapping Filter is to override Quality of Service settings.

Initially, Mapping Filters are empty and will always return true. Hence, we must add constraints by using `'CosNotifyFilter_MappingFilter':add_mapping_constraints/2`. If a constraint matches, the associated value will be used instead of the related Quality of Service system settings.

As input, the second argument must be a sequence of:

```
#'CosNotifyFilter_MappingConstraintPair'{
    constraint_expression = #'CosNotifyFilter_ConstraintExp'{
       event_types = [#'CosNotification_EventType'{
                           domain_name = string(),
                           type_name = string()}],
       constraint_expr = string()},
    result_to_set = any()}
```

1.6 Quality Of Service and Admin Properties

1.6.1 Quality Of Service and Admin Properties

This chapter explains the allowed properties for *CosNotification_QoSAdmin* and *CosNotification_AdminPropertiesAdmin*.

Quality Of Service

The cosNotification application supports the following QoS settings:

QoS	*Range*	*Default*
EventReliability	BestEffort/Persistent	BestEffort
ConnectionReliability	BestEffort/Persistent	BestEffort
Priority	+/-32767	0
OrderPolicy	Any-, Fifo-, Priority- and Deadline-Order	PriorityOrder
DiscardPolicy	RejectNewEvents, Any-, Fifo-, Lifo-, Priority- and Deadline-Order	RejectNewEvents
MaximumBatchSize	long() > 0	1
PacingInterval	TimeBase::TimeT (see cosTime)	0
StartTimeSupported	boolean	false
StopTimeSupported	boolean	false
MaxEventsPerConsumer	long() > 0	100
Timeout	TimeBase::TimeT (see cosTime)	No timeout

Table 6.1: Table 1: Supported QoS Settings

Comments on the table 'Supported QoS Settings':

EventReliability

To allow full Persistent EventReliability, every event must be stored in a stable storage which would create a relatively huge overhead. Hence, only lightweight version of the Persistent QoS is supported. The configuration parameters `max_events`, `interval_events` and `timeout_events` determine the behavior of this setting.

ConnectionReliability

If this QoS is set to BestEffort and a client object returns anything other than `ok` to its associated Proxy, the Proxy will discard all events and terminate. Using Persistent and anything other than `ok` is returned, events will be dropped but the proxy will retry later when next delivery is due. A child may not have Persistent while its parent has BestEffort QoS set, e.g., Proxy vs. Admin. If `OBJECT_NOT_EXIST`, `NO_PERMISSION` or `CosEventComm_Disconnected` is thrown, the associated object will terminate even if this parameter is set to Persistent.

Priority
: This QoS will treat all events as if they have the Priority equal to current value, unless the event itself contains a Priority setting, this event will be treated accordingly. Note: for this property to have any effect, the DiscardPolicy and/or OrderPolicy must be set to PriorityOrder.

OrderPolicy
: If set to PriorityOrder, events with the highest Priority will be delivered first. Deadline order will forward events with shortest expiry time first. If two events have the same priority, they will be delivered in FIFO-order.

DiscardPolicy
: If set to PriorityOrder and MaxEventsPerConsumer limit is reached, events with the lowest Priority will be discarded first. Deadline order will discard events with shortest expiry time first.

MaximumBatchSize
: Only valid if the object is supposed to handle a sequence of structured events and determines the largest amount of events that may be passed each time.

PacingInterval
: Determines how long an object will wait before forwarding a structured event sequence of length equal to, or less than MaximumBatchSize. If set to 0, which is the default behavior, no timeout is used and the events are forwarded when the MaximumBatchSize is reached.

StartTimeSupported
: If set to true events which contains the QoS Property `StartTime` (TimeBase::UtcT - absolute time) will not be delivered until the StartTime value have been exceeded. See also the `cosTime` application.

StopTimeSupported
: If set to true, events which contain the QoS Properties `StopTime` (TimeBase::UtcT - absolute time) or `Timeout` (TimeBase::TimeT - relative time) will be discarded if the object has not been able to deliver the event in time. See also the `cosTime` application.

MaxEventsPerConsumer
: The maximum number of events the associated object may store before discarding events in the way described by the DiscardPolicy.

Timeout
: If this QoS property is not included in the event, and the Property `StopTimeSupported` equals true, this setting will be applied if events cannot be delivered within its time limit.

> **Warning:**
>
> Several of the above QoS Properties can be changed during run-time but we strongly advice not to since, if a relatively large amount of events are waiting for delivery, some of the QoS settings would require a total reorder of the events. The QoS property `ConnectioReliability` may *never* be updated during run-time since it may cause deadlock. Run-time, in this case, means activating the Channel by sending the first event.

Setting Quality Of Service

Assume we have a Consumer Admin object which we want to change the current Quality of Service. Typical usage:

```
QoSPersistent =
   [#'CosNotification_Property'
    {name='CosNotification':'ConnectionReliability'(),
     value=any:create(orber_tc:short(),
     'CosNotification':'Persistent'())}],
'CosNotification_QoSAdmin':set_qos(Ch, QoSPersistent),
```

If it is not possible to set the requested QoS the `UnsupportedQoS` exception is raised, which includes a sequence of `PropertyError`'s describing which QoS, possible range and why is not allowed. The error codes are:

- UNSUPPORTED_PROPERTY - QoS not supported for this type of target object.
- UNAVAILABLE_PROPERTY - due to current QoS settings the given property is not allowed.
- UNSUPPORTED_VALUE - property value out of range; valid range is returned.
- UNAVAILABLE_VALUE - due to current QoS settings the given value is not allowed; valid range is returned.
- BAD_PROPERTY - unrecognized property.
- BAD_TYPE - type of supplied property is incorrect.
- BAD_VALUE - illegal value.

The CosNotification_QoSAdmin interface also supports an operation called `validate_qos/2`. The purpose of this operations is to check if a QoS setting is supported by the target object and if so, the operation returns additional properties which could be optionally added as well.

Admin Properties

The cosNotification application supports the following Admin Properties:

Property	*Range*	*Default*
MaxQueueLength	0	0
MaxConsumers	long() >= 0	0
MaxSuppliers	long() >= 0	0

Table 6.2: Table 2: Supported Admin Properties

According to the OMG specification the default values for Admin Properties is supposed to be `0`, which means that no limit applies to these properties.

> **Note:**
>
> Admin Properties can only be set on a Channel Object level, i.e., they will not have an impact on any Admin or Proxy Objects. Currently, setting the Admin Property `MaxQueueLength` have no effect since we cannot discard events accordingly to the Quality of Service Property `DiscardPolicy`.

1.7 cosNotification Examples

1.7.1 A Tutorial on How to Create a Simple Service

Interface Design

To use the cosNotification application *clients* must be implemented. There are twelve types of clients:

- Structured Push Consumer
- Sequence Push Consumer
- Any Push Consumer
- Structured Pull Consumer

- Sequence Pull Consumer
- Any Pull Consumer
- Structured Push Supplier
- Sequence Push Supplier
- Any Push Supplier
- Structured Pull Supplier
- Sequence Pull Supplier
- Any Pull Supplier

The interfaces for these participants are defined in *CosNotification.idl* and *CosNotifyComm.idl*.

Generating a Client Interface

We start by creating an interface which inherits from the correct interface, e.g., *CosNotifyComm::SequencePushConsumer*. Hence, we must also implement all operations defined in the SequencePushConsumer interface. The IDL-file could look like:

```
#ifndef _MYCLIENT_IDL
#define _MYCLIENT_IDL
#include <CosNotification.idl>
#include <CosNotifyComm.idl>

module myClientImpl {

  interface ownInterface:CosNotifyComm::SequencePushConsumer {

    void ownFunctions(in any NeededArguments)
       raises(Systemexceptions,OwnExceptions);

  };
};

#endif
```

Run the IDL compiler on this file by calling the `ic:gen/1` function. This will produce the API named `myClientImpl_ownInterface.erl`. After generating the API stubs and the server skeletons it is time to implement the servers and if no special options are sent to the IDl compiler the file name is `myClientImpl_ownInterface_impl.erl`.

The callback module must contain the necessary functions inherited from *CosNotification.idl* and *CosNotifyComm.idl*.

How to Run Everything

Below is a short transcript on how to run cosNotification.

```
%% Start Mnesia and Orber
mnesia:delete_schema([node()]),
mnesia:create_schema([node()]),
orber:install([node()]),
mnesia:start(),
orber:start(),

%% If cosEvent not installed before it is necessary to do it now.
cosEventApp:install(),
```

```
%% Install cosNotification in the IFR.
cosNotificationApp:install(30),

%% Register the application specific Client implementations
%% in the IFR.
'oe_myClientImpl':'oe_register'(),

%% Start the cosNotification application.
cosNotificationApp:start(),

%% Start a factory using the default configuration
ChFac = cosNotificationApp:start_factory(),
%% ... or use configuration parameters.
ChFac = cosNotificationApp:start_factory([]),

%% Create a new event channel. Note, if no QoS- anr/or Admin-properties
%% are supplied (i.e. empty list) the default settings are used.
{Ch, ChID} = 'CosNotifyChannelAdmin_EventChannelFactory':
          create_channel(ChFac, DefaultQoS, DefaultAdmin),

%% Retrieve a SupplierAdmin and a Consumer Admin.
{AdminSupplier, ASID}=
         'CosNotifyChannelAdmin_EventChannel':new_for_suppliers(Ch, 'OR_OP'),
{AdminConsumer, ACID}=
         'CosNotifyChannelAdmin_EventChannel':new_for_consumers(Ch,'OR_OP'),

%% Use the corresponding Admin object to get access to wanted Proxies

%% Create a Push Consumer Proxie, i.e., the Client Push Supplier will
%% push events to this Proxy.
{StructuredProxyPushConsumer,ID11}= 'CosNotifyChannelAdmin_SupplierAdmin':
     obtain_notification_push_consumer(AdminSupplier, 'STRUCTURED_EVENT')),

%% Create Push Suppliers Proxies, i.e., the Proxy will push events to the
%% registered Push Consumers.
{ProxyPushSupplier,I4D}='CosNotifyChannelAdmin_ConsumerAdmin':
      obtain_notification_push_supplier(AdminConsumer, 'ANY_EVENT'),
{StructuredProxyPushSupplier,ID5}='CosNotifyChannelAdmin_ConsumerAdmin':
      obtain_notification_push_supplier(AdminConsumer, 'STRUCTURED_EVENT'),
{SequenceProxyPushSupplier,ID6}='CosNotifyChannelAdmin_ConsumerAdmin':
      obtain_notification_push_supplier(AdminConsumer, 'SEQUENCE_EVENT'),

%% Create application Clients. We can, for example, start the Clients
%% our selves or look them up in the naming service. This is application
%% specific.
SupplierClient  = ...
ConsumerClient1 = ...
ConsumerClient2 = ...
ConsumerClient3 = ...

%% Connect each Client to corresponding Proxy.
'CosNotifyChannelAdmin_StructuredProxyPushConsumer':
  connect_structured_push_supplier(StructuredProxyPushConsumer, SupplierClient),
'CosNotifyChannelAdmin_ProxyPushSupplier':
  connect_any_push_consumer(ProxyPushSupplier, ConsumerClient1),
'CosNotifyChannelAdmin_StructuredProxyPushSupplier':
  connect_structured_push_consumer(StructuredProxyPushSupplier, ConsumerClient2),
'CosNotifyChannelAdmin_SequenceProxyPushSupplier':
  connect_sequence_push_consumer(SequenceProxyPushSupplier, ConsumerClient3),
```

The example above, exemplifies a notification system where the SupplierClient in some way generates event and pushes them to the proxy. The push supplier proxies will eventually push the events to each ConsumerClient.

2 Reference Manual

The *cosNotification* application is an Erlang implementation of the OMG CORBA Notification Service.

cosNotificationApp

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module contains the functions for starting and stopping the application.

Exports

`install() -> Return`

Types:

Return = ok | {'EXCEPTION', E}

This operation installs the cosNotification application.

`install(Seconds) -> Return`

Types:

Return = ok | {'EXCEPTION', E}

This operation installs the cosNotification application using `Seconds` delay between each block, currently 6, of IFR-registrations. This approach spreads the IFR database access over a period of time to allow other applications to run smother.

`install_event() -> Return`

Types:

Return = ok | {'EXCEPTION', E}

This operation, which may *only* be used if it is impossible to upgrade to *cosEvent-2.0* or later, installs the necessary cosEvent interfaces. If cosEvent-2.0 is available, use `cosEventApp:install()` instead.

`install_event(Seconds) -> Return`

Types:

Return = ok | {'EXCEPTION', E}

This operation, which may *only* be used if it is impossible to upgrade to *cosEvent-2.0* or later, installs the necessary cosEvent interfaces using `Seconds` delay between each block of IFR-registrations. If cosEvent-2.0 is available, use `cosEventApp:install()` instead.

`uninstall() -> Return`

Types:

Return = ok | {'EXCEPTION', E}

This operation uninstalls the cosNotification application.

`uninstall(Seconds) -> Return`

Types:

Return = ok | {'EXCEPTION', E}

This operation uninstalls the cosNotification application using `Seconds` delay between each block, currently 6, of IFR-unregistrations. This approach spreads the IFR database access over a period of time to allow other applications to run smother.

`uninstall_event() -> Return`

Types:

Return = ok | {'EXCEPTION', E}

This operation uninstalls the inherited cosEvent interfaces. If cosEvent is in use this function may not be used. This function may only be used if `cosNotificationApp:install_event/1/2` was used. If not, use `cosEventApp:uninstall()` instead.

`uninstall_event(Seconds) -> Return`

Types:

Return = ok | {'EXCEPTION', E}

This operation uninstalls the inherited cosEvent interfaces, using `Seconds` delay between each block of IFR-unregistrations. If cosEvent is in use this function may not be used. This function may only be used if `cosNotificationApp:install_event/1/2` was used. If not, use `cosEventApp:uninstall()` instead.

`start() -> Return`

Types:

Return = ok | {error, Reason}

This operation starts the cosNotification application.

`stop() -> Return`

Types:

Return = ok | {error, Reason}

This operation stops the cosNotification application.

`start_global_factory() -> ChannelFactory`

Types:

ChannelFactory = #objref

This operation creates a *Event Channel Factory* should be used for a multi-node Orber. The Factory is used to create a new *channel*.

`start_global_factory(Options) -> ChannelFactory`

Types:

Options = [Option]

Option = {pullInterval, Seconds} | {filterOp, Op} | {gcTime, Seconds} | {gcLimit, Anount} | {timeService, #objref}

ChannelFactory = #objref

This operation creates a *Event Channel Factory* and should be used for a multi-node Orber. The Factory is used to create a new *channel*.

- `{pullInterval, Seconds}` - determine how often Proxy Pull Consumers will check for new events with the client application. The default value is 20 seconds.
- `{filterOp, OperationType}` - determine which type of Administrator objects should be started, i.e., `'OR_OP'` or `'AND_OP'`. The default value is `'OR_OP'`.
- `{timeService, TimeServiceObj | 'undefined'}` - to be able to use Start and/or Stop QoS this option must be used. See the function `start_time_service/2` in the `cosTime` application. The default value is `'undefined'`.
- `{filterOp, OperationType}` - determine which type of Administrator objects should be started, i.e., `'OR_OP'` or `'AND_OP'`. The default value is `'OR_OP'`.
- `{gcTime, Seconds}` - this option determines how often, for example, proxies will garbage collect expired events. The default value is 60.
- `{gcLimit, Amount}` - determines how many events will be stored before, for example, proxies will garbage collect expired events. The default value is 50. This option is tightly coupled with the QoS property `MaxEventsPerConsumer`, i.e., the `gcLimit` should be less than `MaxEventsPerConsumer` and greater than 0.

`start_factory() -> ChannelFactory`

Types:

ChannelFactory = #objref

This operation creates a *Event Channel Factory*. The Factory is used to create a new *channel*.

`start_factory(Options) -> ChannelFactory`

Types:

Options = [Option]

Option = {pullInterval, Seconds} | {filterOp, Op} | {gcTime, Seconds} | {gcLimit, Amount} | {timeService, #objref}

ChannelFactory = #objref

This operation creates a *Event Channel Factory*. The Factory is used to create a new *channel*.

`stop_factory(ChannelFactory) -> Reply`

Types:

ChannelFactory = #objref

Reply = ok | {'EXCEPTION', E}

This operation stop the target channel factory.

`start_filter_factory() -> FilterFactory`

Types:

FilterFactory = #objref

This operation creates a *Filter Factory*. The Factory is used to create a new *Filter's* and *MappingFilter's*.

`stop_filter_factory(FilterFactory) -> Reply`

Types:

FilterFactory = #objref

Reply = ok | {'EXCEPTION', E}

This operation stop the target filter factory.

`create_structured_event(Domain, Type, Event, VariableHeader, FilterableBody, BodyRemainder) -> Reply`

Types:

Domain = string()

Type = string()

Event = string()

VariableHeader = [CosNotification::Property]

FilterableBody = [CosNotification::Property]

BodyRemainder = #any data-type

Reply = CosNotification::StructuredEvent | {'EXCEPTION', E}

An easy way to create a structured event is to use this function. Simple typechecks are performed and if one of the arguments is not correct a 'BAD_PARAM' exception is thrown.

`type_check() -> Reply`

Types:

Reply = true | false

This operation returns the value of the configuration parameter `type_check`.

CosNotifyChannelAdmin_EventChannelFactory

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

Exports

`create_channel(ChannelFactory, InitialQoS, InitialAdmin) -> Return`

Types:

ChannelFactory = #objref

InitialQoS = CosNotification::QoSProperties

InitialAdmin = CosNotification::AdminProperties

Return = {EventChannel, ChannelID}

EventChannel = #objref

ChannelID = long()

This operation creates a new event channel. Along with the channel reference an id is returned which can be used when invoking other operations exported by this module. The Quality of Service argument supplied will be inherited by objects created by the channel. For more information about QoS settings see the `User's Guide`.

If no QoS- and/or Admin-properties are supplied (i.e. empty list), the *default* settings will be used. For more information, see the User's Guide.

`get_all_channels(ChannelFactory) -> ChannelIDSeq`

Types:

ChannelFactory = #objref

ChannelIDSeq = [long()]

This operation returns a id sequence of all channel's created by this ChannelFactory.

`get_event_channel(ChannelFactory, ChannelID) -> Return`

Types:

ChannelFactory = #objref

ChannelID = long()

Retrurn = EventChannel | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

EventChannel = #objref

This operation returns the EventChannel associated with the given id. If no channel is associated with the id, i.e., never existed or have been terminated, an exception is raised.

CosNotifyChannelAdmin_EventChannel

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotification_QoSAdmin*
- *CosNotification_AdminPropertiesAdmin*

Exports

`_get_MyFactory(Channel) -> ChannelFactory`

Types:

Channel = #objref

ChannelFactory = #objref

This readonly attribute maintains the reference of the event channel factory that created the target channel.

`_get_default_consumer_admin(Channel) -> ConsumerAdmin`

Types:

Channel = #objref

ConsumerAdmin = #objref

This is a readonly attribute which maintains a reference to a default `ConsumerAdmin` object associated with the target object.

`_get_default_supplier_admin(Channel) -> SupplierAdmin`

Types:

Channel = #objref

SupplierAdmin = #objref

This is a readonly attribute which maintains a reference to a default `SupplierAdmin` object associated with the target object.

`_get_default_filter_factory(Channel) -> FilterFactory`

Types:

Channel = #objref

FilterFactory = #objref

The default `FilterFactory` associated with the target channel is maintained by this readonly attribute.

`new_for_consumers(Channel, OpType) -> Return`

Types:

Channel = #objref

OpType = 'AND_OP' | 'OR_OP'

Return = {ConsumerAdmin, AdminID}

ConsumerAdmin = #objref

AdminID = long()

This operation creates a new instance of a `ConsumerAdmin` and supplies an Id which may be used when invoking other operations exported by this module. The returned object will inherit the Quality of Service properties of the target channel.

`for_consumers(Channel) -> ConsumerAdmin`

Types:

Channel = #objref

ConsumerAdmin = #objref

A new new instance of a `ConsumerAdmin` object is created but no Id is returned. The returned object's operation type, i.e., `'AND_OP'` or `'OR_OP'`, will be set to the value of the configuration parameter `filterOp`. The target object's Quality of Service properties will be inherited by the returned `ConsumerAdmin`.

`new_for_suppliers(Channel, OpType) -> Return`

Types:

Channel = #objref

OpType = 'AND_OP' | 'OR_OP'

Return = {SupplierAdmin, AdminID}

SupplierAdmin = #objref

AdminID = long()

Enables us to create a new instance of a `SupplierAdmin`. An Id, which may be used when invoking other operations exported by this module, is also returned. The current Quality of Service settings associated with the target object will be inherited by the `SupplierAdmin`.

`for_suppliers(Channel) -> SupplierAdmin`

Types:

Channel = #objref

SupplierAdmin = #objref

To create a new `SupplierAdmin` with the target object's current Quality of Service settings we can use this function. The returned object's operation type (`'AND_OP'` or `'OR_OP'`) will be determined by the configuration variable `filterOp`.

`get_consumeradmin(Channel, AdminID) -> ConsumerAdmin`

Types:

Channel = #objref

AdminID = long()

ConsumerAdmin = #objref | {'EXCEPTION', #'CosNotifyChannelAdmin_AdminNotFound'{}}

If the given Id is associated with a `ConsumerAdmin` the object reference is returned. If such association never existed or the `ConsumerAdmin` have terminated an exception is raised.

`get_supplieradmin(Channel, AdminID) -> SupplierAdmin`

Types:

Channel = #objref

AdminID = long()

SupplierAdmin = #objref | {'EXCEPTION', #'CosNotifyChannelAdmin_AdminNotFound'{}}

Equal to the operation `get_consumeradmin/2` but a reference to a `SupplierAdmin` is returned.

`get_all_consumeradmins(Channel) -> Reply`

Types:

Channel = #objref

Reply = [AdminID]

AdminID = long()

To get access to all `ConsumerAdmin` Id's created by the target object, and still alive, this operation could be invoked.

`get_all_supplieradmins(Channel) -> Reply`

Types:

Channel = #objref

Reply = [AdminID]

AdminID = long()

Equal to the operation `get_all_consumeradmins/1` but returns a list of all `SupplierAdmin` object ID's.

`destroy(Channel) -> ok`

Types:

Channel = #objref

The `destroy` operation will terminate the target channel and all associated Admin objects.

CosNotification

Erlang module

To get access to all definitions include necessary `hrl` files by using:
`-include_lib("cosNotification/include/*.hrl").`

Exports

`'EventReliability'() -> string()`

This function returns the EventReliability QoS identifier

`'BestEffort'() -> short()`

This function returns the BestEffort QoS value.

`'Persistent'() -> short()`

This function returns the Persistent QoS value.

`'ConnectionReliability'() -> string()`

This function returns the ConnectionReliability QoS identifier.

`'Priority'() -> string()`

This function returns the Priority QoS identifier.

`'LowestPriority'() -> short()`

This function returns the LowestPriority QoS value.

`'HighestPriority'() -> short()`

This function returns the HighestPriority QoS value.

`'DefaultPriority'() -> short()`

This function returns the DefaultPriority QoS value.

`'StartTime'() -> string()`

This function returns the StartTime QoS identifier.

`'StopTime'() -> string()`

This function returns the StopTime QoS identifier.

`'Timeout'() -> string()`

This function returns the Timeout QoS identifier.

'OrderPolicy'() -> string()

This function returns the OrderPolicy QoS identifier.

'AnyOrder'() -> short()

This function returns the AnyOrder QoS value.

'FifoOrder'() -> short()

This function returns the FifoOrder QoS value.

'PriorityOrder'() -> short()

This function returns the PriorityOrder QoS value.

'DeadlineOrder'() -> short()

This function returns the DeadlineOrder QoS value.

'DiscardPolicy'() -> string()

This function returns the DiscardPolicy QoS identifier.

'LifoOrder'() -> short()

This function returns the LifoOrder QoS value.

'RejectNewEvents'() -> short()

This function returns the RejectNewEvents QoS value.

'MaximumBatchSize'() -> string()

This function returns the MaximumBatchSize QoS identifier.

'PacingInterval'() -> string()

This function returns the PacingInterval QoS identifier.

'StartTimeSupported'() -> string()

This function returns the StartTimeSupported QoS identifier.

'StopTimeSupported'() -> string()

This function returns the StopTimeSupported QoS identifier.

'MaxEventsPerConsumer'() -> string()

This function returns the MaxEventsPerConsumer QoS identifier.

'MaxQueueLength'() -> string()

This function returns the MaxQueueLength Admin identifier.

'MaxConsumers'() -> string()

This function returns the MaxConsumers Admin identifier.

'MaxSuppliers'() -> string()

This function returns the MaxSuppliers Admin identifier.

CosNotification_QoSAdmin

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

All objects, which inherit this interface, export functions described in this module.

Exports

`get_qos(Object) -> Reply`

Types:

Object = #objref

Reply = [QoSProperty]

QoSProperty = #'CosNotification_Property'{name, value}

name = string()

value = #any

This operation returns a list of name-value pairs which encapsulates the current QoS settings for the target object.

`set_qos(Object, QoS) -> Reply`

Types:

Object = #objref

QoS = [QoSProperty]

QoSProperty = #'CosNotification_Property'{name, value}

name = string()

value = #any

Reply = ok | {'EXCEPTION', #'CosNotification_UnsupportedQoS'{qos_err}}

qos_err = PropertyErrorSeq

PropertyErrorSeq = [PropertyError]

PropertyError = #'CosNotification_PropertyError'{code, name, available_range}

code = 'UNSUPPORTED_PROPERTY' | 'UNAVAILABLE_PROPERTY' | 'UNSUPPORTED_VALUE' | 'UNAVAILABLE_VALUE' | 'BAD_PROPERTY' | 'BAD_TYPE' | 'BAD_VALUE'

name = string()

available_range = PropertyRange

PropertyRange = #CosNotification_PropertyRange{low_val, high_val}

low_val = high_val = #any

To alter the current QoS settings for the target object this function must be used. If it is not possible to set the requested QoS the `UnsupportedQoS` exception is raised, which includes a sequence of `PropertyError`'s describing which QoS, possible range and why is not allowed.

`validate_qos(Object, QoS) -> Reply`

Types:

Object = #objref

QoS = [QoSProperty]

QoSProperty = #'Property'{name, value}
name = string()
value = #any
Reply = {ok, NamedPropertyRangeSeq} | {'EXCEPTION', CosNotification_UnsupportedQoS{}}
NamedPropertyRangeSeq = [NamedPropertyRange]
NamedPropertyRange = #CosNotification_NamedPropertyRange{name, range}
name = string()
range = #CosNotification_PropertyRange{low_val, high_val}
low_val = #any
high_val = #any

The purpose of this operations is to check if a QoS setting is supported by the target object and if so, the operation returns additional properties which could be optionally added as well.

CosNotification_AdminPropertiesAdmin

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

All objects, which inherit this interface, export functions described in this module.

Exports

`get_admin(Object) -> AdminProperties`

Types:

Object = #objref

AdminProperties = [AdminProperty]

AdminProperty = #'CosNotification_Property'{name, value}

name = string()

value = #any

This operation returns sequence of name-value pairs which encapsulates the current administrative properties of the target object.

`set_admin(Object, AdminProperties) -> Reply`

Types:

Object = #objref

AdminProperties = [AdminProperty]

AdminProperty = #'CosNotification_Property'{name, value}

name = string()

value = #any

Reply = ok | {'EXCEPTION', CosNotification_UnsupportedAdmin}

As input, this operation accepts a sequence of name-value pairs encapsulating the desired administrative settings for the target object. If it is not possible to set the given properties the exception `UnsupportedAdmin` will be raised.

CosNotifyChannelAdmin_ConsumerAdmin

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotification_QoSAdmin*
- *CosNotifyComm_NotifySubscribe*
- *CosNotifyFilter_FilterAdmin*

Exports

`_get_MyID(ConsumerAdmin) -> AdminID`

Types:

ConsumerAdmin = #objref

AdminID = long()

The ID returned by the creating channel is equal to the value encapsulated by this readonly attribute.

`_get_MyChannel(ConsumerAdmin) -> Channel`

Types:

ConsumerAdmin = #objref

Channel = #objref

The creating channel's reference is maintained by this readonly attribute.

`_get_MyOperator(ConsumerAdmin) -> OpType`

Types:

ConsumerAdmin = #objref

OpType = 'AND_OP' | 'OR_OP'

When `ConsumerAdmin's` are created an operation type, i.e., `'AND_OP'` or `'OR_OP'`, is supplied, which determines the semantics used by the target object concerning evaluation against any associated `Filter` objects.

`_get_priority_filter(ConsumerAdmin) -> MappingFilter`

Types:

ConsumerAdmin = MappingFilter = #objref

If set, this operation returns the associated priority `MappingFilter`, otherwise a `NIL` object reference is returned.

`_set_priority_filter(ConsumerAdmin, MappingFilter) -> ok`

Types:

ConsumerAdmin = MappingFilter = #objref

To associate a priority `MappingFilter` with the target object this operation must be used.

`_get_lifetime_filter(ConsumerAdmin) -> MappingFilter`

Types:

ConsumerAdmin = MappingFilter = #objref

Unless a lifetime `MappingFilter` have been associated with the target object a `NIL` object reference is returned by this operation.

`_set_lifetime_filter(ConsumerAdmin, MappingFilter) -> ok`

Types:

ConsumerAdmin = MappingFilter = #objref

This operation associate a lifetime `MappingFilter` with the target object.

`_get_pull_suppliers(ConsumerAdmin) -> ProxyIDSeq`

Types:

ConsumerAdmin = #objref

ProxyIDSeq = [ProxyID]

ProxyID = long()

This readonly attribute maintains the Id's for all `PullProxies` created by the target object and still alive.

`_get_push_suppliers(ConsumerAdmin) -> ProxyIDSeq`

Types:

ConsumerAdmin = #objref

ProxyIDSeq = [ProxyID]

ProxyID = long()

This attribute is similar to the `_get_pull_suppliers` attribute but maintains the Id's for all `PushProxies` created by the target object and still alive.

`get_proxy_supplier(ConsumerAdmin, ProxyID) -> Reply`

Types:

ConsumerAdmin = #objref

ProxyID = long()

Reply = Proxy | {'EXCEPTION', #'CosNotifyChannelAdmin_ProxyNotFound'{}}

Proxy = #objref

If a proxy with the given Id exists the reference to the object is returned, but if the object have terminated, or an incorrect Id is supplied, an exception is raised.

`obtain_notification_pull_supplier(ConsumerAdmin, ConsumerType) -> Reply`

Types:

ConsumerAdmin = #objref

ConsumerType = 'ANY_EVENT' | 'STRUCTURED_EVENT' | 'SEQUENCE_EVENT'

Reply = {Proxy, ProxyID}

Proxy = #objref

ProxyID = long()

Determined by the parameter `ConsumerType`, a proxy which will accept events of the defined type is created. Along with the object reference an Id is returned.

`obtain_pull_supplier(ConsumerAdmin) -> Proxy`

Types:

ConsumerAdmin = #objref

Proxy = #objref

This operation creates a new proxy which accepts `#any{}` events.

`obtain_notification_push_supplier(ConsumerAdmin, ConsumerType) -> Reply`

Types:

ConsumerAdmin = #objref

ConsumerType = 'ANY_EVENT' | 'STRUCTURED_EVENT' | 'SEQUENCE_EVENT'

Reply = {Proxy, ProxyID}

Proxy = #objref

ProxyID = long()

A proxy which accepts events of the type described by the parameter `ConsumerType` is created by this operation. A unique Id is returned as an out parameter.

`obtain_push_supplier(ConsumerAdmin) -> Proxy`

Types:

ConsumerAdmin = #objref

Proxy = #objref

The object created by this function is a proxy which accepts `#any{}` events.

`destroy(ConsumerAdmin) -> ok`

Types:

ConsumerAdmin = #objref

To terminate the target object this operation should be used. The associated `Channel` will be notified.

CosNotifyChannelAdmin_SupplierAdmin

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotification_QoSAdmin*
- *CosNotifyComm_NotifyPublish*
- *CosNotifyFilter_FilterAdmin*

Exports

`_get_MyID(SupplierAdmin) -> AdminID`

Types:

SupplierAdmin = #objref

AdminID = long()

When a `SupplierAdmin` object is created it is given a unique Id by the creating channel. This readonly attribute maintains this Id.

`_get_MyChannel(SupplierAdmin) -> Channel`

Types:

SupplierAdmin = #objref

Channel = #objref

The creating channel's reference is maintained by this readonly attribute.

`_get_MyOperator(SupplierAdmin) -> OpType`

Types:

SupplierAdmin = #objref

OpType = 'AND_OP' | 'OR_OP'

The Operation Type, which determines the semantics the target object will use for any associated `Filters`, is maintained by this readonly attribute.

`_get_pull_consumers(SupplierAdmin) -> ProxyIDSeq`

Types:

SupplierAdmin = #objref

ProxyIDSeq = [ProxyID]

ProxyID = long()

A sequence of all associated `PullProxy` Id's is maintained by this readonly attribute.

`_get_push_consumers(SupplierAdmin) -> ProxyIDSeq`

Types:

SupplierAdmin = #objref

ProxyIDSeq = [ProxyID]

ProxyID = long()

This operation returns all `PushProxy` Id's created by the target object.

`get_proxy_consumer(SupplierAdmin, ProxyID) -> Reply`

Types:

SupplierAdmin = #objref

ProxyID = long()

Reply = Proxy | {'EXCEPTION', #'CosNotifyChannelAdmin_ProxyNotFound'{}}

Proxy = #objref

The Proxy which corresponds to the given Id is returned by this operation.

`obtain_notification_pull_consumer(SupplierAdmin, SupplierType) -> Reply`

Types:

SupplierAdmin = #objref

SupplierType = 'ANY_EVENT' | 'STRUCTURED_EVENT' | 'SEQUENCE_EVENT'

Reply = {Proxy, ProxyID}

Proxy = #objref

ProxyID = long()

This operation creates a new proxy and returns its object reference along with its ID. The `SupplierType` parameter determines the event type accepted by the proxy.

`obtain_pull_consumer(SupplierAdmin) -> Proxy`

Types:

SupplierAdmin = #objref

Proxy = #objref

A proxy which accepts `#any{}` events is created by this operation.

`obtain_notification_push_consumer(SupplierAdmin, SupplierType) -> Reply`

Types:

SupplierAdmin = #objref

SupplierType = 'ANY_EVENT' | 'STRUCTURED_EVENT' | 'SEQUENCE_EVENT'

Reply = {Proxy, ProxyID}

Proxy = #objref

ProxyID = long()

Determined by the `SupplierType` parameter a compliant proxy is created and its object reference along with its Id is returned by this operation.

`obtain_push_consumer(SupplierAdmin) -> Proxy`

Types:

SupplierAdmin = #objref

Proxy = #objref

A proxy which accepts `#any{}` events is created by this operation.

`destroy(SupplierAdmin) -> ok`

Types:

SupplierAdmin = #objref

This operation terminates the SupplierAdmin object and notifies the creating channel that the target object no longer is active.

CosNotifyComm_NotifyPublish

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

All objects, which inherit this interface, export functions described in this module.

Exports

`offer_change(Object, Added, Removed) -> Reply`

Types:

Object = #objref

Added = Removed = EventTypeSeq

EventTypeSeq = [type]

Reply = ok | {'EXCEPTION', CosNotifyComm_InvalidEventType{type}}

type = #'CosNotification_EventType'{domain_name, type_name}

domain_name = type_name = string()

Objects supporting this interface can be informed by supplier objects about which type of events that will be delivered in the future. This operation accepts two parameters describing new and old event types respectively. If any of the supplied event type names is syntactically incorrect an exception is raised.

CosNotifyComm_NotifySubscribe

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

All objects, which inherit this interface, export functions described in this module.

Exports

`subscription_change(Object, Added, Removed) -> Reply`

Types:

Object = #objref

Added = Removed = EventTypeSeq

EventTypeSeq = [type]

Reply = ok | {'EXCEPTION', CosNotifyComm_InvalidEventType{type}}

type = #'CosNotification_EventType'{domain_name, type_name}

domain_name = type_name = string()

This operation takes as input two sequences of event type names specifying events the client will and will not accept in the future respectively.

CosNotifyFilter_FilterAdmin

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

All objects, which inherit this interface, export functions described in this module.

Exports

`add_filter(Object, Filter) -> FilterID`

Types:

Object = #objref

Filter = #objref

FilterID = long()

This operation connects a new `Filter` to the target object. This `Filter` will, together with other associated `Filters`, be used to select events to forward. A unique Id is returned and should be used if we no longer want to consult the given `Filter`.

`remove_filter(Object, FilterID) -> ok`

Types:

Object = #objref

FilterID = long()

If a certain `Filter` no longer should be associated with the target object this operation must be used. Events will no longer be tested against the `Filter` associated with the given Id.

`get_filter(Object, FilterID) -> Reply`

Types:

Object = #objref

FilterID = long()

Reply = Filter | {'EXCEPTION', #'CosNotifyFilter_FilterNotFound'{}}

Filter = #objref

If the target object is associated with a `Filter` matching the given Id the reference will be returned. If no such `Filter` is known by the target object an exception is raised.

`get_all_filters(Object) -> FilterIDSeq`

Types:

Object = #objref

FilterIDSeq = [FilterID]

FilterID = long()

Id's for all `Filter` objects associated with the target object is returned by this operation.

`remove_all_filters(Object) -> ok`

Types:

Object = #objref

If we want to remove all `Filters` associated with the target object we can use this function.

CosNotifyFilter_FilterFactory

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

Exports

`create_filter(FilterFactory, Grammar) -> Reply`

Types:

FilterFactory = #objref

Grammar = string()

Reply = Filter | {'EXCEPTION', #'CosNotifyFilter_InvalidGrammar'{}}

Filter = #objref

This operation creates a new Filter object, under the condition that Grammar given is supported. Currently, only `"EXTENDED_TCL"` is supported.

`create_mapping_filter(FilterFactory, Grammar) -> Reply`

Types:

FilterFactory = #objref

Grammar = string()

Reply = MappingFilter | {'EXCEPTION', #'CosNotifyFilter_InvalidGrammar'{}}

Filter = #objref

This operation creates a new MappingFilter object, under the condition that Grammar given is supported. Currently, only `"EXTENDED_TCL"` is supported.

CosNotifyFilter_Filter

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

Exports

`_get_constraint_grammar(Filter) -> Grammar`

Types:

Filter = #objref

Grammar = string()

This operation returns which type of Grammar the Filter uses. Currently, only `"EXTENDED_TCL"` is supported.

`add_constraints(Filter, ConstraintExpSeq) -> Reply`

Types:

Filter = #objref

ConstraintExpSeq = [Constraint]

ConstraintExp = #'CosNotifyFilter_ConstraintExp'{event_types, constraint_expr}

event_types = #'CosNotification_EventTypeSeq'{}

constraint_expr = string()

Reply = ConstraintInfoSeq | {'EXCEPTION', #'CosNotifyFilter_InvalidConstraint'{constr}}

constr = ConstraintExp

ConstraintInfoSeq = [ConstraintInfo]

ConstraintInfo = #'CosNotifyFilter_ConstraintInfo'{constraint_expression, constraint_id}

constraint_expression = ConstraintExp

constraint_id = long()

Initially, Filters do not contain any constraints, hence, all events will be forwarded. The `add_constraints/2` operation allow us to add constraints to the target object.

`modify_constraints(Filter, ConstraintIDSeq, ConstraintInfoSeq) -> Reply`

Types:

Filter = #objref

ConstraintIDSeq = [ConstraintID]

ConstraintID = long()

ConstraintInfoSeq = [ConstraintInfo]

ConstraintInfo = #'CosNotifyFilter_ConstraintInfo'{constraint_expression, constraint_id}

constraint_expression = ConstraintExp

constraint_id = long()

Reply = ok | {'EXCEPTION', #'CosNotifyFilter_InvalidConstraint'{constr}} | {'EXCEPTION', #'CosNotifyFilter_ConstraintNotFound'{id}}

constr = ConstraintExp

id = long()

ConstraintExp = #'CosNotifyFilter_ConstraintExp'{event_types, constraint_expr}

event_types = #'CosNotification_EventTypeSeq'{}

constraint_expr = string()

This operation is invoked by a client in order to modify the constraints associated with the target object. The constraints related to the Id's in the parameter sequence `ConstraintIDSeq` will, if all values are valid, be deleted. The `ConstraintInfoSeq` parameter contains of Id-Expression pairs and a constraint matching one of the unique Id's will, if all input values are correct, be updated. If the parameters contain incorrect data en exception will be raised.

`get_constraints(Filter, ConstraintIDSeq) -> Reply`

Types:

Filter = #objref

ConstraintIDSeq = [ConstraintID]

ConstraintID = long()

Reply = ConstraintInfoSeq | {'EXCEPTION', #'CosNotifyFilter_ConstraintNotFound'{id}}

ConstraintInfoSeq = [ConstraintInfo]

ConstraintInfo = #'CosNotifyFilter_ConstraintInfo'{constraint_expression, constraint_id}

constraint_expression = ConstraintExp

constraint_id = id = long()

This operation return a sequence of ConstraintInfo's, related to the given ConstraintID's, associated with the target object.

`get_all_constraints(Filter) -> ConstraintInfoSeq`

Types:

Filter = #objref

ConstraintInfoSeq = [ConstraintInfo]

ConstraintInfo = #'CosNotifyFilter_ConstraintInfo'{constraint_expression, constraint_id}

constraint_expression = ConstraintExp

constraint_id = long()

All constraints, and their unique Id, associated with the target object will be returned by this operation.

`remove_all_constraints(Filter) -> ok`

Types:

Filter = #objref

All constraints associated with the target object are removed by this operation and, since the the target object no longer contain any constraints, true will always be the result of any match operation.

`destroy(Filter) -> ok`

Types:

Filter = #objref

This operation terminates the target object.

`match(Filter, Event) -> Reply`

Types:

Filter = #objref

Event = #any

Reply = boolean() | {'EXCEPTION', #'CosNotifyFilter_UnsupportedFilterableData'{}}

This operation accepts an `#any{}` event and returns `true` if it satisfies at least one constraint. If the event contains data of the wrong type, e.g., should be a string() but in fact i a short(), an exception is raised.

`match_structured(Filter, Event) -> Reply`

Types:

Filter = #objref

Event = #'CosNotification_StructuredEvent'{}

Reply = boolean() | {'EXCEPTION', #'CosNotifyFilter_UnsupportedFilterableData'{}}

This operation is similar to the `match` operation but accepts structured events instead.

`attach_callback(Filter, NotifySubscribe) -> CallbackID`

Types:

Filter = #objref

NotifySubscribe = #objref

CallbackID = long()

This operation connects a NotifySubscribe object, which should be informed when the target object's constraints are updated. A unique Id is returned which must be stored if we ever want to detach the callback object in the future.

`detach_callback(Filter, CallbackID) -> Reply`

Types:

Filter = #objref

CallbackID = long()

Reply = ok | {'EXCEPTION', #'CosNotifyFilter_CallbackNotFound'{}}

If the target object has an associated callback that matches the supplied Id it will be removed and longer informed of any updates. If no object with a matching Id is found an exception is raised.

`get_callbacks(Filter) -> CallbackIDSeq`

Types:

Filter = #objref

CallbackIDSeq = [CallbackID]

CallbackID = long()

This operation returns a sequence of all connected NotifySubscribe object Id's. If no callbacks are associated with the target object the list will be empty.

CosNotifyFilter_MappingFilter

Erlang module

The main purpose of this module is to match events against associated constraints and return the value for the first constraint that returns true for the given event. If all constraints return false the default value will be returned.

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

Exports

`_get_constraint_grammar(MappingFilter) -> Grammar`

Types:

MappingFilter = #objref

Grammar = string()

This operation returns which type of Grammar the MappingFilter uses. Currently, only `"EXTENDED_TCL"` is supported.

`_get_value_type(MappingFilter) -> CORBA::TypeCode`

Types:

MappingFilter = #objref

This readonly attribute maintains the `CORBA::TypeCode` of the default value associated with the target object.

`_get_default_value(MappingFilter) -> #any`

Types:

MappingFilter = #objref

This readonly attribute maintains the `#any{}` default value associated with the target object.

`add_mapping_constraints(MappingFilter, MappingConstraintPairSeq) -> Reply`

Types:

MappingFilter = #objref

MappingConstraintPairSeq = [MappingConstraintPair]

MappingConstraintPair = #'CosNotifyFilter_MappingConstraintPair'{constraint_expression, result_to_set}

constraint_expression = #'CosNotifyFilter_ConstraintExp'{event_types, constraint_expr}

event_types = #'CosNotification_EventTypeSeq'{}

constraint_expr = string()

result_to_set = #any

Reply = MappingConstraintInfoSeq | {'EXCEPTION', #'CosNotifyFilter_InvalidConstraint'{constr}} | {'EXCEPTION', #'CosNotifyFilter_InvalidValue'{constr, value}}

constr = ConstraintExp

ConstraintExp = #'CosNotifyFilter_ConstraintExp'{event_types, constraint_expr}

event_types = #'CosNotification_EventTypeSeq'{}

constraint_expr = string()

MappingConstraintInfoSeq = [MappingConstraintInfo]

MappingConstraintInfo = #'CosNotifyFilter_MappingConstraintInfo'{constraint_expression, constraint_id, value}

constraint_expression = ConstraintExp

constraint_id = long()

value = #any

This operation add new mapping constraints, which will be used when trying to override Quality of Service settings defined in the given event. If a constraint return true the associated value will be returned, otherwise the default value.

`modify_constraints(MappingFilter, ConstraintIDSeq, MappingConstraintInfoSeq) -> Reply`

Types:

MappingFilter = #objref

ConstraintIDSeq = [ConstraintID]

ConstraintID = long()

MappingConstraintInfoSeq = [MappingConstraintInfo]

MappingConstraintInfo = #'CosNotifyFilter_MappingConstraintInfo'{constraint_expression, constraint_id, value}

constraint_expression = ConstraintExp

constraint_id = long()

value = #any

ConstraintInfoSeq = [ConstraintInfo]

ConstraintInfo = #'CosNotifyFilter_ConstraintInfo'{constraint_expression, constraint_id}

constraint_expression = ConstraintExp

constraint_id = long()

Reply = ok | {'EXCEPTION', #'CosNotifyFilter_InvalidConstraint'{constr}} | {'EXCEPTION', #'CosNotifyFilter_ConstraintNotFound'{id}} | {'EXCEPTION', #'CosNotifyFilter_InvalidValue'{constr, value}}

constr = ConstraintExp

id = long()

value = #any

ConstraintExp = #'CosNotifyFilter_ConstraintExp'{event_types, constraint_expr}

event_types = #'CosNotification_EventTypeSeq'{}

constraint_expr = string()

The `ConstraintIDSeq` supplied should relate to constraints the caller wishes to remove. If any of the supplied Id's are not found an exception will be raised. This operation also accepts a sequence of `MappingConstraintInfo` which will be added. If the target object cannot modify the constraints as requested an exception is raised describing which constraint, and why, could not be updated.

`get_mapping_constraints(MappingFilter, ConstraintIDSeq) -> Reply`

Types:

MappingFilter = #objref

ConstraintIDSeq = [ConstraintID]

ConstraintID = long()

Reply = MappingConstraintInfoSeq | {'EXCEPTION', #'CosNotifyFilter_ConstraintNotFound'{id}}

MappingConstraintInfoSeq = [MappingConstraintInfo]

MappingConstraintInfo = #'CosNotifyFilter_MappingConstraintInfo'{constraint_expression, constraint_id, value}

constraint_expression = ConstraintExp

ConstraintExp = #'CosNotifyFilter_ConstraintExp'{event_types, constraint_expr}

event_types = #'CosNotification_EventTypeSeq'{}

constraint_expr = string()

constraint_id = id = long()

value = #any

When adding a new constraint a unique Id is returned, which is accepted as input for this operation. The associated constraint is returned, but if no such Id exists an exception is raised.

`get_all_mapping_constraints(MappingFilter) -> MappingConstraintInfoSeq`

Types:

MappingFilter = #objref

MappingConstraintInfoSeq = [MappingConstraintInfo]

MappingConstraintInfo = #'CosNotifyFilter_MappingConstraintInfo'{constraint_expression, constraint_id, value}

constraint_expression = ConstraintExp

ConstraintExp = #'CosNotifyFilter_ConstraintExp'{event_types, constraint_expr}

event_types = #'CosNotification_EventTypeSeq'{}

constraint_expr = string()

constraint_id = long()

value = #any

This operation returns a sequence of all unique Id's associated with the target object. If no constraint have been added the sequence will be empty.

`remove_all_mapping_constraints(MappingFilter) -> ok`

Types:

MappingFilter = #objref

This operation removes all constraints associated with the target object.

`destroy(MappingFilter) -> ok`

Types:

MappingFilter = #objref

This operation terminates the target object. Remember to remove this Filter from the objects it have been associated with.

`match(MappingFilter, Event) -> Reply`

Types:

MappingFilter = #objref

Event = #any

Reply = {boolean(), #any} | {'EXCEPTION', #'CosNotifyFilter_UnsupportedFilterableData'{}}

This operation evaluates `Any` events with the Filter's constraints, and returns the value to use. The value is the default value if all constraints returns false and the value associated with the first constraint returning true.

`match_structured(MappingFilter, Event) -> Reply`

Types:

MappingFilter = #objref

Event = #'CosNotification_StructuredEvent'{}

Reply = {boolean(), #any} | {'EXCEPTION', #'CosNotifyFilter_UnsupportedFilterableData'{}}

Similar to `match/2` but accepts a structured event as input.

CosNotifyChannelAdmin_ProxyConsumer

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*

Exports

`_get_MyType(ProxyConsumer) -> ProxyType`

Types:

ProxyConsumer = #objref

ProxyType = 'PUSH_ANY' | 'PULL_ANY' | 'PUSH_STRUCTURED' | 'PULL_STRUCTURED' | 'PUSH_SEQUENCE' | 'PULL_SEQUENCE'

This readonly attribute maintains the enumerant describing the which type the target object is.

`_get_MyAdmin(ProxyConsumer) -> AdminObject`

Types:

ProxyConsumer = AdminObject = #objref

This readonly attribute maintains the admin's reference which created the target object.

`obtain_subscription_types(ProxyConsumer, ObtainInfoMode) -> EventTypeSeq`

Types:

ProxyConsumer = #objref

ObtainInfoMode = 'ALL_NOW_UPDATES_OFF' | 'ALL_NOW_UPDATES_ON' | 'NONE_NOW_UPDATES_OFF' | 'NONE_NOW_UPDATES_ON'

EventTypeSeq = [EventType]

EventType = #'CosNotification_EventType'{domain_name, type_name}

domain_name = type_name = string()

Depending on the input parameter `ObtainInfoMode`, this operation may return a sequence of the `EventTypes` the target object is interested in receiving. If `'ALL_NOW_UPDATES_OFF'` or `'ALL_NOW_UPDATES_ON'` is given a sequence will be returned, otherwise not. If `'ALL_NOW_UPDATES_OFF'` or `'NONE_NOW_UPDATES_OFF'` are issued the target object will not inform the associated `NotifySubscribe` object when an update occurs. `'ALL_NOW_UPDATES_ON'` or `'NONE_NOW_UPDATES_ON'` will result in that update information will be sent.

`validate_event_qos(ProxyConsumer, QoSProperties) -> Reply`

Types:

ProxyConsumer = #objref

QoSProperties = [QoSProperty]

QoSProperty = #'CosNotification_Property'{name, value}

name = string()

value = #any
Reply = {ok, NamedPropertyRangeSeq} | {'EXCEPTION', CosNotification_UnsupportedQoS{qos_err}}
NamedPropertyRangeSeq = [NamedPropertyRange]
NamedPropertyRange = #CosNotification_NamedPropertyRange{name, range}
name = string()
range = #CosNotification_PropertyRange{low_val, high_val}
low_val = #any
high_val = #any
qos_err = PropertyErrorSeq
PropertyErrorSeq = [PropertyError]
PropertyError = #'CosNotification_PropertyError'{code, name, available_range}
code = 'UNSUPPORTED_PROPERTY' | 'UNAVAILABLE_PROPERTY' | 'UNSUPPORTED_VALUE' | 'UNAVAILABLE_VALUE' | 'BAD_PROPERTY' | 'BAD_TYPE' | 'BAD_VALUE'
name = string()
available_range = PropertyRange
PropertyRange = #CosNotification_PropertyRange{low_val, high_val}
low_val = high_val = #any

To check if certain Quality of Service properties can be added to events in the current context of the target object this operation should be used. If we cannot support the required settings an exception describing why will be raised.

CosNotifyChannelAdmin_ProxySupplier

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*

Exports

`_get_MyType(ProxySupplier) -> ProxyType`

Types:

ProxySupplier = #objref

ProxyType = 'PUSH_ANY' | 'PULL_ANY' | 'PUSH_STRUCTURED' | 'PULL_STRUCTURED' | 'PUSH_SEQUENCE' | 'PULL_SEQUENCE'

This readonly attribute maintains the enumerant describing the which type the target object is.

`_get_MyAdmin(ProxySupplier) -> AdminObject`

Types:

ProxySupplier = #objref

AdminObject = #objref

This readonly attribute maintains the admin's reference which created the target object.

`_get_priority_filter(ProxySupplier) -> MappingFilter`

Types:

ProxySupplier = #objref

MappingFilter = #objref

This operation returns the associated priority MappingFilter. If no such object exist a `NIL` reference is returned.

`_set_priority_filter(ProxySupplier, MappingFilter) -> ok`

Types:

ProxySupplier = #objref

MappingFilter = #objref

This operation associate a new priority MappingFilter with the target object.

`_get_lifetime_filter(ProxySupplier) -> MappingFilter`

Types:

ProxySupplier = #objref

MappingFilter = #objref

This operation returns the associated lifetime MappingFilter. If no such object exist a `NIL` reference is returned.

`_set_lifetime_filter(ProxySupplier, MappingFilter) -> ok`

Types:

ProxySupplier = #objref

MappingFilter = #objref

This operation associate a new lifetime MappingFilter with the target object.

`obtain_offered_types(ProxySupplier, ObtainInfoMode) -> EventTypeSeq`

Types:

ProxySupplier = #objref

ObtainInfoMode = 'ALL_NOW_UPDATES_OFF' | 'ALL_NOW_UPDATES_ON' | 'NONE_NOW_UPDATES_OFF' | 'NONE_NOW_UPDATES_ON'

EventTypeSeq = [EventType]

EventType = #'CosNotification_EventType'{domain_name, type_name}

domain_name = type_name = string()

Depending on the input parameter `ObtainInfoMode`, this operation may return a sequence of the `EventTypes` the target object is interested in receiving. If `'ALL_NOW_UPDATES_OFF'` or `'ALL_NOW_UPDATES_ON'` is given a sequence will be returned, otherwise not. If `'ALL_NOW_UPDATES_OFF'` or `'NONE_NOW_UPDATES_OFF'` are issued the target object will not inform the associated `NotifySubscribe` object when an update occurs. `'ALL_NOW_UPDATES_ON'` or `'NONE_NOW_UPDATES_ON'` will result in that update information will be sent.

`validate_event_qos(ProxySupplier, QoSProperties) -> Reply`

Types:

ProxySupplier = #objref

QoSProperties = [QoSProperty]

QoSProperty = #'CosNotification_Property'{name, value}

name = string()

value = #any

Reply = {ok, NamedPropertyRangeSeq} | {'EXCEPTION', CosNotification_UnsupportedQoS{qos_err}}

NamedPropertyRangeSeq = [NamedPropertyRange]

NamedPropertyRange = #CosNotification_NamedPropertyRange{name, range}

name = string()

range = #CosNotification_PropertyRange{low_val, high_val}

low_val = #any

high_val = #any

qos_err = PropertyErrorSeq

PropertyErrorSeq = [PropertyError]

PropertyError = #'CosNotification_PropertyError'{code, name, available_range}

code = 'UNSUPPORTED_PROPERTY' | 'UNAVAILABLE_PROPERTY' | 'UNSUPPORTED_VALUE' | 'UNAVAILABLE_VALUE' | 'BAD_PROPERTY' | 'BAD_TYPE' | 'BAD_VALUE'

name = string()

available_range = PropertyRange

PropertyRange = #CosNotification_PropertyRange{low_val, high_val}

low_val = high_val = #any

To check if certain Quality of Service properties can be added to events in the current context of the target object this operation should be used. If we cannot support the required settings an exception describing why will be raised.

CosNotifyChannelAdmin_ProxyPullConsumer

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifyPublish*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxyConsumer*

Exports

`connect_any_pull_supplier(ProxyPullConsumer, PullSupplier) -> Reply`

Types:

ProxyPullConsumer = #objref

PullSupplier = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}} | {'EXCEPTION', #'CosEventChannelAdmin_TypeError'{}}

This operation connects the given `PullSupplier` to the target object. If a client is already connected the `AlreadyConnected` exception will be raised. The client must support the operations `pull` and `try_pull`, otherwise the `TypeError` exception is raised.

`suspend_connection(ProxyPullConsumer) -> Reply`

Types:

ProxyPullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

If we want to temporarily suspend the connection with the target object this operation must be sued. If the connection already have been suspended or no client have been connected an exception is raised.

`resume_connection(ProxyPullConsumer) -> Reply`

Types:

ProxyPullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyActive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

If The connection have been suspended earlier we can invoke this operation to reinstate the connection. If the connection already is active or no client have been connected to the target object an exception is raised.

`disconnect_pull_consumer(ProxyPullConsumer) -> ok`

Types:

ProxyPullConsumer = #objref

Invoking this operation disconnects the client from the target object which then terminates and inform its administrative parent.

CosNotifyChannelAdmin_ProxyPullSupplier

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifySubscribe*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxySupplier*

Exports

`connect_any_pull_consumer(ProxyPullSupplier, PullConsumer) -> Reply`

Types:

ProxyPullSupplier = #objref

PullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}}

This operation connects the given `PullConsumer` to the target object. If a connection already exists the `AlreadyConnected` exception is raised.

`pull(ProxyPullSupplier) -> Reply`

Types:

ProxyPullSupplier = #objref

Reply = #any | {'EXCEPTION', #'CosEventChannelAdmin_Disconnected'{}}

This operation pulls next `#any{}` event, and blocks, if the target object have no events to forward, until an event can be delivered. If no client have been connected the `Disconnected` exception is raised.

`try_pull(ProxyPullSupplier) -> Reply`

Types:

ProxyPullSupplier = #objref

Reply = {#any, HasEvent} | {'EXCEPTION', #'CosEventChannelAdmin_Disconnected'{}}

HasEvent = boolean()

This operation pulls next event, but do not block if the target object have no event to forward. If no client have been connected the `Disconnected` exception is raised.

`disconnect_pull_supplier(ProxyPullSupplier) -> ok`

Types:

ProxyPullSupplier = #objref

Invoking this operation will cause the target object to close the connection and terminate.

CosNotifyChannelAdmin_ProxyPushConsumer

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifyPublish*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxyConsumer*

Exports

`connect_any_push_supplier(ProxyPushConsumer, PushSupplier) -> Reply`

Types:

ProxyPushConsumer = #objref

PushSupplier = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}}

This operation connects a `PushSupplier` to the target object. If a connection already exists the `AlreadyConnected` exception is raised.

`push(ProxyPushConsumer, Event) -> Reply`

Types:

ProxyPushConsumer = #objref

Event = #any

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_Disconnected'{}}

This operation pushes an `#any{}` event to the target object. If no client have been connected the `Disconnected` exception is raised.

`disconnect_push_consumer(ProxyPushConsumer) -> ok`

Types:

ProxyPushConsumer = #objref

Invoking this operation will cause the target object to close the connection and terminate.

CosNotifyChannelAdmin_ProxyPushSupplier

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifySubscribe*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmi*
- *CosNotifyChannelAdmin_ProxySupplier*

Exports

`connect_any_push_consumer(ProxyPushSupplier, PushConsumer) -> Reply`

Types:

ProxyPushSupplier = #objref

PushConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}} | {'EXCEPTION', #'CosEventChannelAdmin_TypeError'{}}

This operation connects a `PushConsumer` to the target object. If a connection already exists or the given client does not support the operation `push` an exception, `AlreadyConnected` and `TypeError` respectively, is raised.

`suspend_connection(ProxyPushSupplier) -> Reply`

Types:

ProxyPushSupplier = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

This operation suspends the connection with the client object. If the connection already is suspended or no client have been associated an exception is raised.

`resume_connection(ProxyPushSupplier) -> Reply`

Types:

ProxyPullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

If a connection have been suspended earlier, calling this operation will resume the connection. If the connection already is active or no client have been connected an exception is raised.

`disconnect_push_supplier(ProxyPushSupplier) -> ok`

Types:

ProxyPushSupplier = #objref

This operation cause the target object to close the connection and terminate.

CosNotifyChannelAdmin_SequenceProxyPullConsumer

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifyPublish*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxyConsumer*

Exports

`connect_sequence_pull_supplier(SequenceProxyPullConsumer, PullSupplier) -> Reply`

Types:

SequenceProxyPullConsumer = #objref

PullSupplier = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}} | {'EXCEPTION', #'CosEventChannelAdmin_TypeError'{}}

This operation connects a `PullSupplier` to the target object. If a connection already exists or the supplied client does not support the functions `pull_structured_events` and `try_pull_structured_events` an exception is raised.

`suspend_connection(SequenceProxyPullConsumer) -> Reply`

Types:

SequenceProxyPullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

If a connection exist, invoking this operation will suspend the connection until instructed otherwise. Otherwise, no client have been connected or this operation already have been invoked an exception is raised.

`resume_connection(SequenceProxyPullConsumer) -> Reply`

Types:

SequenceProxyPullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

If an connection have been suspended this operation must be used to resume the connection. If the connection already is active or no client have been connected an exception is raised.

`disconnect_sequence_pull_consumer(SequenceProxyPullConsumer) -> ok`

Types:

SequenceProxyPullConsumer = #objref

This operation close the connection to the client and terminates the target object.

CosNotifyChannelAdmin_SequenceProxyPullSupplier

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifySubscribe*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxySupplier*

Exports

`connect_sequence_pull_consumer(SequenceProxyPullSupplier, PullConsumer) -> Reply`

Types:

SequenceProxyPullSupplier = #objref

PullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}}

This operation connects a `PullConsumer` to the target object. If a connection already exists an exception is raised.

`pull_structured_events(SequenceProxyPullSupplier, MaxEvents) -> Reply`

Types:

SequenceProxyPullSupplier = #objref

MaxEvents = long()

Reply = EventBatch | {'EXCEPTION', #'CosEventChannelAdmin_Disconnected'{}}

EventBatch = [StructuredEvent]

StructuredEvent = #'CosNotification_StructuredEvent'{header, filterable_data, remainder_of_body}

header = EventHeader

filterable_data = [#'CosNotification_Property'{name, value}]

name = string()

value = #any

remainder_of_body = #any

EventHeader = #'CosNotification_EventHeader'{fixed_header, variable_header}

fixed_header = FixedEventHeader

variable_header = OptionalHeaderFields

FixedEventHeader = #'CosNotification_FixedEventHeader'{event_type, event_name}

event_type = EventType

event_name = string()

EventType = #'CosNotification_EventType'{domain_name, type_name}

domain_name = type_name = string()

OptionalHeaderFields = [#'CosNotification_Property'{name, value}]

A client use this operation to pull next event sequence of maximum length `MaxEvents`. This operation is blocking and will not reply until the requested amount of events can be delivered or the QoS property `PacingInterval` is reached. For more information see the `User's Guide`.

`try_pull_structured_events(SequenceProxyPullSupplier, MaxEvents) -> Reply`

Types:

SequenceProxyPullSupplier = #objref
MaxEvents = long()
Reply = {EventBatch, HasEvent} | {'EXCEPTION', #'CosEventChannelAdmin_Disconnected'{}}
HasEvent = boolean()
EventBatch = [StructuredEvent]
StructuredEvent = #'CosNotification_StructuredEvent'{header, filterable_data, remainder_of_body}
header = EventHeader
filterable_data = [#'CosNotification_Property'{name, value}]
name = string()
value = #any
remainder_of_body = #any
EventHeader = #'CosNotification_EventHeader'{fixed_header, variable_header}
fixed_header = FixedEventHeader
variable_header = OptionalHeaderFields
FixedEventHeader = #'CosNotification_FixedEventHeader'{event_type, event_name}
event_type = EventType
event_name = string()
EventType = #'CosNotification_EventType'{domain_name, type_name}
domain_name = type_name = string()
OptionalHeaderFields = [#'CosNotification_Property'{name, value}]

This operation pulls an event sequence of the maximum length `MaxEvents`, but do not block if the target object have no events to forward. The outparameter, `HasEvent` is true if the sequence contain any events.

`disconnect_sequence_pull_supplier(SequenceProxyPullSupplier) -> ok`

Types:

SequenceProxyPullSupplier = #objref

This operation cause the target object to close the connection and terminate.

CosNotifyChannelAdmin_SequenceProxyPushConsumer

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifyPublish*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxyConsumer*

Exports

`connect_sequence_push_supplier(SequenceProxyPushConsumer, PushSupplier) -> Reply`

Types:

SequenceProxyPushConsumer = #objref

PushSupplier = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}}

This operation connects a `PushSupplier` to the target object. If a connection already exists the `AlreadyConnected` exception is raised.

`push_structured_events(SequenceProxyPushConsumer, EventBatch) -> Reply`

Types:

SequenceProxyPushConsumer = #objref

EventBatch = [StructuredEvent]

StructuredEvent = #'CosNotification_StructuredEvent'{header, filterable_data, remainder_of_body}

header = EventHeader

filterable_data = [#'CosNotification_Property'{name, value}]

name = string()

value = #any

remainder_of_body = #any

EventHeader = #'CosNotification_EventHeader'{fixed_header, variable_header}

fixed_header = FixedEventHeader

variable_header = OptionalHeaderFields

FixedEventHeader = #'CosNotification_FixedEventHeader'{event_type, event_name}

event_type = EventType

event_name = string()

EventType = #'CosNotification_EventType'{domain_name, type_name}

domain_name = type_name = string()

OptionalHeaderFields = [#'CosNotification_Property'{name, value}]

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_Disconnected'{}}

A client must use this operation when it wishes to push a new sequence of events to the target object. If no connection exists the `Disconnected` exception is raised.

`disconnect_sequence_push_consumer(SequenceProxyPushConsumer) -> ok`

Types:

SequenceProxyPushConsumer = #objref

This operation cause the target object to close the connection and terminate.

CosNotifyChannelAdmin_SequenceProxyPushSupplier

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifySubscribe*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxySupplier*

Exports

`connect_sequence_push_consumer(SequenceProxyPushSupplier, PushConsumer) -> Reply`

Types:

SequenceProxyPushSupplier = #objref

PushConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}} | {'EXCEPTION', #'CosEventChannelAdmin_TypeError'{}}

This operation connects a `PushConsumer` to the target object. If a connection already exists or the function `psuh_structured_events` is not supported the exceptions `AlreadyConnected` or `TypeError` will be raised respectively.

`suspend_connection(SequenceProxyPushSupplier) -> Reply`

Types:

SequenceProxyPushSupplier = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

This operation suspends the connection between the client and the target object. If no connection exists or the connection is already suspended an exception is raised.

`resume_connection(SequenceProxyPushSupplier) -> Reply`

Types:

SequenceProxyPullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

If the connection have previously been suspended this operation must used if we want to resume the connection. If no object have been connected or the connection already is active an exception is raised.

`disconnect_sequence_push_supplier(SequenceProxyPushSupplier) -> ok`

Types:

SequenceProxyPushSupplier = #objref

This operation cause the target object to close the connection and terminate.

CosNotifyChannelAdmin_StructuredProxyPullConsumer

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifyPublish*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxyConsumer*

Exports

`connect_structured_pull_supplier(StructuredProxyPullConsumer, PullSupplier) -> Reply`

Types:

StructuredProxyPullConsumer = #objref

PullSupplier = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}} | {'EXCEPTION', #'CosEventChannelAdmin_TypeError'{}}

This operation connects a `PullSupplier` to the target object. If a connection already exists or the given client object does not support the functions `pull_structured_event` and `try_pull_structured_event` an exception is raised.

`suspend_connection(StructuredProxyPullConsumer) -> Reply`

Types:

StructuredProxyPullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

This operation suspends the connection between the target object and its client. If no connection exists or already suspended an exception is raised.

`resume_connection(StructuredProxyPullConsumer) -> Reply`

Types:

StructuredProxyPullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

If the connection have been suspended this operation must be used if we want to resume the connection. If the connection already are active or no connection have been created an exception is raised.

`disconnect_structured_pull_consumer(StructuredProxyPullConsumer) -> ok`

Types:

StructuredProxyPullConsumer = #objref

This operation cause the target object to close the connection and terminate.

CosNotifyChannelAdmin_StructuredProxyPullSupplier

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifySubscribe*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxySupplier*

Exports

`connect_structured_pull_consumer(StructuredProxyPullSupplier, PullConsumer) -> Reply`

Types:

StructuredProxyPullSupplier = #objref

PullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}}

This operation connects a `PullConsumer` to the target object. If a connection already exists the `AlreadyConnected` exception is raised.

`pull_structured_event(StructuredProxyPullSupplier) -> Reply`

Types:

StructuredProxyPullSupplier = #objref

Reply = StructuredEvent | {'EXCEPTION', #'CosEventChannelAdmin_Disconnected'{}}

StructuredEvent = #'CosNotification_StructuredEvent'{header, filterable_data, remainder_of_body}

header = EventHeader

filterable_data = [#'CosNotification_Property'{name, value}]

name = string()

value = #any

remainder_of_body = #any

EventHeader = #'CosNotification_EventHeader'{fixed_header, variable_header}

fixed_header = FixedEventHeader

variable_header = OptionalHeaderFields

FixedEventHeader = #'CosNotification_FixedEventHeader'{event_type, event_name}

event_type = EventType

event_name = string()

EventType = #'CosNotification_EventType'{domain_name, type_name}

domain_name = type_name = string()

OptionalHeaderFields = [#'CosNotification_Property'{name, value}]

This operation pulls next event from the target object; if an event cannot be delivered this function blocks until an event arrives.

`try_pull_structured_event(StructuredProxyPullSupplier) -> Reply`

Types:

StructuredProxyPullSupplier = #objref
Reply = {StructuredEvent, HasEvent} | {'EXCEPTION', #'CosEventChannelAdmin_Disconnected'{}}
HasEvent = boolean()
StructuredEvent = #'CosNotification_StructuredEvent'{header, filterable_data, remainder_of_body}
header = EventHeader
filterable_data = [#'CosNotification_Property'{name, value}]
name = string()
value = #any
remainder_of_body = #any
EventHeader = #'CosNotification_EventHeader'{fixed_header, variable_header}
fixed_header = FixedEventHeader
variable_header = OptionalHeaderFields
FixedEventHeader = #'CosNotification_FixedEventHeader'{event_type, event_name}
event_type = EventType
event_name = string()
EventType = #'CosNotification_EventType'{domain_name, type_name}
domain_name = type_name = string()
OptionalHeaderFields = [#'CosNotification_Property'{name, value}]

This operation try to pull next event from the target object. If no event have arrived an empty event is returned and the out parameter `HasEvent` is set to false. Otherwise, the boolean flag is set to true and an valid event is returned.

`disconnect_structured_pull_supplier(StructuredProxyPullSupplier) -> ok`

Types:

StructuredProxyPullSupplier = #objref

This operation cause the target object to close the connection and terminate.

CosNotifyChannelAdmin_StructuredProxyPushConsumer

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifyPublish*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxyConsumer*

Exports

`connect_structured_push_supplier(StructuredProxyPushConsumer, PushSupplier) -> Reply`

Types:

StructuredProxyPushConsumer = #objref

PushSupplier = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}}

This operation connects a `PushSupplier` to the target object. If a connection already exists an exception is raised.

`push_structured_event(StructuredProxyPushConsumer, StructuredEvent) -> Reply`

Types:

StructuredProxyPushConsumer = #objref

StructuredEvent = #'CosNotification_StructuredEvent'{header, filterable_data, remainder_of_body}

header = EventHeader

filterable_data = [#'CosNotification_Property'{name, value}]

name = string()

value = #any

remainder_of_body = #any

EventHeader = #'CosNotification_EventHeader'{fixed_header, variable_header}

fixed_header = FixedEventHeader

variable_header = OptionalHeaderFields

FixedEventHeader = #'CosNotification_FixedEventHeader'{event_type, event_name}

event_type = EventType

event_name = string()

EventType = #'CosNotification_EventType'{domain_name, type_name}

domain_name = type_name = string()

OptionalHeaderFields = [#'CosNotification_Property'{name, value}]

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_Disconnected'{}}

When a client want to push a new event to the target object this operation must be used.

disconnect_structured_push_consumer(StructuredProxyPushConsumer) -> ok

Types:

StructuredProxyPushConsumer = #objref

This operation cause the target object to close the connection and terminate.

CosNotifyChannelAdmin_StructuredProxyPushSupplier

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosNotification/include/*.hrl").`

This module also exports the functions described in:

- *CosNotifyComm_NotifySubscribe*
- *CosNotification_QoSAdmin*
- *CosNotifyFilter_FilterAdmin*
- *CosNotifyChannelAdmin_ProxySupplier*

Exports

`connect_structured_push_consumer(StructuredProxyPushSupplier, PushConsumer) -> Reply`

Types:

StructuredProxyPushSupplier = #objref

PushConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}} | {'EXCEPTION', #'CosEventChannelAdmin_TypeError'{}}

This operation connects a `PushConsumer` to the target object. If a connection already exists or the function `push_structured_event` is not supported by the client object an exception is raised.

`suspend_connection(StructuredProxyPushSupplier) -> Reply`

Types:

StructuredProxyPushSupplier = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

This operation suspends the connection with the target object. If no connection exists or the connection already is suspended an exception is raised.

`resume_connection(StructuredProxyPushSupplier) -> Reply`

Types:

StructuredProxyPullConsumer = #objref

Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ConnectionAlreadyInactive'{}} | {'EXCEPTION', #'CosNotifyChannelAdmin_NotConnected'{}}

If the connection with the target object have been suspended this function must be used to resume the connection. If no client have been connected or the connection is active an exception is raised.

`disconnect_structured_push_supplier(StructuredProxyPushSupplier) -> ok`

Types:

StructuredProxyPushSupplier = #objref

This operation cause the target object to close the connection and terminate.