cosEvent Application

version 1.0

Helen Airiyan

1998-04-25

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

1 CosEvent User's Guide 1

1.1 The cosEvent Application . . . 2

Content Overview . . . 2

Brief description of the User's Guide . . . 2

1.2 Introduction to cosEvent . . . 3

Overview . . . 3

1.3 Event Service . . . 4

Overview of the CosEvent Service . . . 4

Event Service Components . . . 4

Event Service Communication Models . . . 5

Creating an EventChannel . . . 6

Using the Event Service . . . 6

1.4 cosEvent Release Notes . . . 9

cosEvent 1.0.1, Release Notes . . . 9

cosEvent 1.0, Release Notes . . . 9

2 cosEvent Referens Manual 11

2.1 CosEventChannelAdmin (Module) . . . 14

2.2 CosEventChannelAdmin_ConsumerAdmin (Module) . . . 17

2.3 CosEventChannelAdmin_EventChannel (Module) . . . 18

2.4 CosEventChannelAdmin_ProxyPullConsumer (Module) . . . 20

2.5 CosEventChannelAdmin_ProxyPullSupplier (Module) . . . 21

2.6 CosEventChannelAdmin_ProxyPushConsumer (Module) . . . 23

2.7 CosEventChannelAdmin_ProxyPushSupplier (Module) . . . 25

2.8 CosEventChannelAdmin_SupplierAdmin (Module) . . . 26

2.9 OrberEventChannel (Module) . . . 27

2.10 OrberEventChannel_EventChannelFactory (Module) . . . 28

List of Figures 29

Chapter 1

CosEvent User's Guide

The cosEvent Application is an Erlang implementation of a CORBA Service CosEvent.

1.1 The cosEvent Application

Content Overview

The cosEvent documentation is divided into three sections:

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

Brief description of the User's Guide

The User’s Guide contains the following parts:

- CosEvent overview
- CosEvent installation and examples

1.2 Introduction to cosEvent

Overview

The cosEvent application is a Event Service compliant with the OMG[1] Event Service CosEvent.

Purpose and Dependencies

CosEvent is dependent on *Orber*, which provides CORBA functionality in an Erlang environment.

Prerequisites

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

Recommended reading includes *CORBA, Fundamentals and Programming - Jon Siegel* and *Open Telecom Platform Documentation Set*. It is also helpful to have read *Concurrent Programming in Erlang*.

[1] URL: `http://www.omg.org`

1.3 Event Service

Overview of the CosEvent Service

The Event service allows programmers to subscribe to information channels. Suppliers can generate events without knowing the consumer identities and the consumer can receive events without knowing the supplier identity. Both push and pull event delivery are supported. The Event service will queue information and processes.

The CORBA Event service provides a flexible model for asynchronous, decoupled communication between objects. This chapter outlines communication models and the roles and relationships of key components in the CosEvent service. It shows a simple example on use of this service.

Event Service Components

There are five components in the OMG CosEvent service architecture. These are described below:

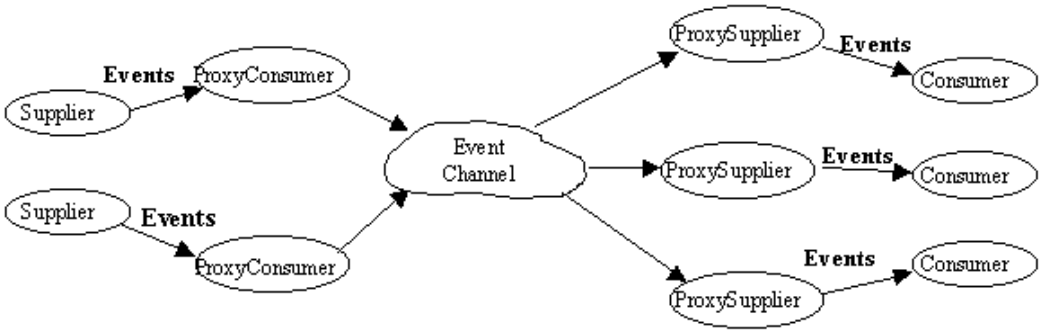

Figure 1.1: Figure 1: Event service Components

- *Suppliers and consumers:* Consumers are the ultimate targets of events generated by suppliers. Consumers and suppliers can both play active and passive roles. There could be two types of consumers and suppliers: push or pull. A PushSupplier object can actively push an event to a passive PushConsumer object. Likewise, a PullSupplier object can passively wait for a PullConsumer object to actively pull an event from it.
- *EventChannel:* The central abstraction in the CosEvent service is the EventChannel which plays the role of a mediator between consumers and suppliers. Consumers and suppliers register their interest with the EventChannel. It can provide many-to-many communication. The channel consumes events from one or more suppliers, and supplies events to one or more consumers. An EventChannel can support consumers and suppliers using different communication models.
- *ProxySuppliers and ProxyConsumers:* ProxySuppliers act as middlemen between consumers and the EventChannel. A ProxySupplier is similar to a normal supplier, but includes an additional method for connecting a consumer to the ProxySupplier. Likewise, ProxyConsumers act as

middlemen between suppliers and the EventChannel. A ProxyConsumer is similar to a normal consumer, but includes an additional method for connecting a supplier to the ProxyConsumer.

- *Supplier and consumer administrations:* Consumer administration acts as a factory for creating ProxySuppliers. Supplier administration acts as a factory for creating ProxyConsumers.

Event Service Communication Models

There are four general models of component collaboration in the OMG CosEvent service architecture. The following describes these models: (Please note that proxies are not shown in the diagrams for simplicity).

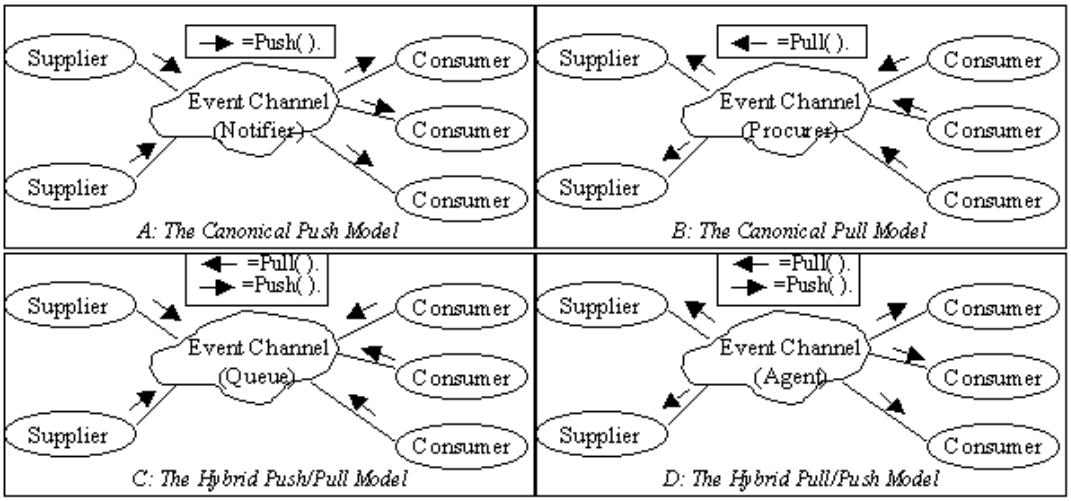

Figure 1.2: Figure 2: Event service Communication Models

- *The Canonical Push Model:* The Canonical push model shown in figure 2(A) allows the suppliers of events to initiate the transfer of event data to consumers. In this model, suppliers are active initiators and consumers are the passive targets of the requests. EventChannels play the role of `Notifier`. Thus, active suppliers use EventChannels to push data to passive consumers that have registered with the EventChannels.
- *The Canonical Pull Model:*The Canonical pull model shown in figure 2(B) allows consumers to request events from suppliers. In this model, Consumers are active initiators and suppliers are the passive targets of the pull requests. EventChannel plays the role of `Procurer` since it procures events on behalf of consumers. Thus, active consumers can explicitly pull data from passive suppliers via the EventChannels.
- *The Hybrid Push/Pull Model:* The push/pull model shown in figure 2(C) is a hybrid that allows consumers to request events queued at an EventChannel by suppliers. In this model, both suppliers and consumers are active initiators of the requests. EventChannels play the role of `Queue`. Thus, active consumers can explicitly pull data deposited by active suppliers via the EventChannels.

- *The Hybrid Pull/Push Model:* The pull/push model shown in figure 2(D) is another hybrid that allows the channel to pull events from suppliers and push them to consumers. In this model, suppliers are passive targets of pull requests and consumers are passive targets of pushes. EventChannels play the role of `Intelligent Agent`. Thus, active EventChannels can pull data from passive suppliers and push that data to passive consumers.

Creating an EventChannel

An EventChannel can be created by using the `EventChannelFactory` interface, which is implemented by `OrberEventChannel_EventChannelFactory`.

To start the factory server one needs to make a call to `corba:create/2` which could look like this:

```
-module(event_channel_factory).

-include_lib("orber/include/corba.hrl").
-include_lib("orber/COSS/CosNaming/CosNaming.hrl").
-include_lib("orber/COSS/CosNaming/lname.hrl").

-export([start/0]).

start() ->
  ECFok = 'OrberEventChannel_EventChannelFactory':oe_create(),
  NS = corba:resolve_initial_references("Nameservice"),
  NC = lname_component:set_id(lname_component:create(),
                              "EventChannelFactory"),
  N = lname:insert_component(lname:create(), 1, NC),
  'CosNaming_NamingContext':bind(NS, N, ECFok).
```

Now an EventChannelFactory is created and registered in the CosNaming service and could be found by consumers and suppliers.

Using the Event Service

This section shows an example of usage of the Event service in order to decouple communication between a measurements collector and a safety controller.

Using the Consumer interface for safety controller

The safety controller plays the role of a PushConsumer. It is interested in the data provided by the measurements collector, which plays the role of a PushSupplier. Safety controller is responsible for the action required in case some measurements exceed the safety limits.

First, the safety controller creates a PushConsumer itself, and then obtains an EventSupplier channel object reference using the `EventChannelFactory`, as follows:

```
// The safety controller creates a PushConsumer object
MyPushConsumer = my_push_consumer_srv:create(),

// EventChannel created through EventChannelFactory
// EventChannelFactory obtained from the CosNaming service (not shown)
// EventChannel registered in the CosNaming service (not shown)
EventChannel = 'OrberEventChannel_EventChannelFactory':
                create_event_channel(ECFactory),
```

This code assumes that the `MyPushConsumer` supports the `PushConsumer` interface and implements the appropriate safety controller logic.

Note: If no support exists for the push consumer the process will crash.

Next, the safety controller connects itself to the `EventChannel`:

```
// first step: obtain ConsumerAdmin object reference
ConsumerAdmin = 'CosEventChannelAdmin_EventChannel'
                 :for_consumers(EventChannel),
// obtain ProxyPushSupplier from the ConsumerAdmin object
PPhS = 'CosEventChannelAdmin_ConsumerAdmin'
        :obtain_push_supplier(ConsumerAdmin),

// second step: connect our PushConsumer to the ProxyPushSupplier
'CosEventChannelAdmin_ProxyPushSupplier'
         :connect_push_consumer(PPhS, MyPushConsumer)
```

When an event arrives in the EventChannel, it will invoke the `push` operation on the registered `PushConsumer` object reference.

Using the supplier interface for measurements collector

Measurements collector sends data containing information about current measurement of the system to the EventChannel in order to keep safety controller informed of any changes.

As with the safety controller, the measurements collector needs an object reference to an `EventChannel` and to a PushSupplier to connect to the channel. This is accomplished as follows:

```
// measurements collector creates a PushSupplier
MyPushSupplier = my_push_supplier_srv:create(),

// EventChannel obtained from the Naming service (not shown)
EventChannel = //...

// obtain SupplierAdmiin object reference
SupplierAdmin =
  'CosEventChannelAdmin_EventChannel':for_suppliers(EventChannel),

// obtain ProxyPushConsumer from SupplierAdmin object
PPhC =
  'CosEventChannelAdmin_SupplierAdmin':obtain_push_consumer(SupplierAdmin),

// connect our PushSupplier to the ProxyPushConsumer
'CosEventChannelAdmin_ProxyPushConsumer':connect_push_supplier(PPhC,
                                                    MyPushSupplier),
```

Once the consumer and the supplier registration code get executed, both the safety controller and the measurements collector are connected to the EventChannel. At this point, safety controller will automatically receive measurements data that are pushed by the measurements collector.

Exchanging and processing event data

The events exchanged between supplier and consumer must always be specified in OMG IDL so that they can be stored into an any type variable. Consider the following data example sent by the measurements controller:

```
record(measurements, {temperature, pressure, water_level}).
```

In order to push an event, the measurements collector must create and initialize this record, put it into CORBA::any, and call push on the EventChannel PushConsumer interface:

```
// create some data
EventRecord = #measurements{temperature = 150, pressure = 100,
                                        water_level = 200},
EventData = { measurements:tc(),EventRecord},

// push the event to consumer
'CosEventChannelAdmin_ProxyPushConsumer':push(PPhC, EventData),
```

Once the EventChannel receives an event from the measurements collector, it pushes the event data to the consumer by invoking the push operation on registered PushConsumer object reference.

The implementation of the safety controller consumer push could look like this:

```
push(Data) ->
{
  if
    Data#measurements.temperature > 300 ->
      // some logic to set alarm
      ;
    Data#measurements.water_level < 50 ->
      // some logic to get more water
      ;

    ......etc
  end.
}
```

1.4 cosEvent Release Notes

cosEvent 1.0.1, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

-

Incompatibilities

- CosEvent is now able to handle upgrade properly.

Known bugs and problems

-

cosEvent 1.0, Release Notes

Improvements and new features

-

Fixed bugs and malfunctions

-

Incompatibilities

- CosEvent include paths have changed since it is now a separate application, called cosEvent, i.e., no longer a Orber sub-application.

Known bugs and problems

-

cosEvent Referens Manual

Short Summaries

- Erlang Module CosEventChannelAdmin [page ??] – The CosEventChannelAdmin defines a set if event service interfaces that enables decoupled asynchronous communication between objects and implements generic (untyped) version of the OMG COSS standard event service.
- Erlang Module CosEventChannelAdmin_ConsumerAdmin [page ??] – This module implements a ConsumerAdmin interface, which allows consumers to be connected to the event channel.
- Erlang Module CosEventChannelAdmin_EventChannel [page ??] – This module implements an Event Channel interface, which plays the role of a mediator between consumers and suppliers.
- Erlang Module CosEventChannelAdmin_ProxyPullConsumer [page ??] – This module implements a ProxyPullConsumer interface which acts as a middleman between pull supplier and the event channel.
- Erlang Module CosEventChannelAdmin_ProxyPullSupplier [page ??] – This module implements a ProxyPullSupplier interface which acts as a middleman between pull consumer and the event channel.
- Erlang Module CosEventChannelAdmin_ProxyPushConsumer [page ??] – This module implements a ProxyPushConsumer interface which acts as a middleman between push supplier and the event channel.
- Erlang Module CosEventChannelAdmin_ProxyPushSupplier [page ??] – This module implements a ProxyPushSupplier interface which acts as a middleman between push consumer and the event channel.
- Erlang Module CosEventChannelAdmin_SupplierAdmin [page ??] – This module implements a SupplierAdmin interface, which allows suppliers to be connected to the event channel.
- Erlang Module OrberEventChannel [page ??] – The OrberEventChannel defines an interface that enables the user to create event channel objects.
- Erlang Module OrberEventChannel_EventChannelFactory [page ??] – This module implements the EventChannelFactory interface that enables the user to create event channel objects.

CosEventChannelAdmin

No functions are exported

CosEventChannelAdmin_ConsumerAdmin

The following functions are exported:

- `obtain_push_supplier(Object) -> Return`
  [page 17] Creates a ProxyPushSupplier object
- `obtain_pull_supplier(Object) -> Return`
  [page 17] Creates a ProxyPullSupplier object

CosEventChannelAdmin_EventChannel

The following functions are exported:

- `for_consumers(Object) -> Return`
  [page 18] Returns a ConsumerAdmin object
- `for_suppliers(Object) -> Return`
  [page 18] Returns a SupplierAdmin object
- `destroy(Object) -> Return`
  [page 18] Destroys the event channel

CosEventChannelAdmin_ProxyPullConsumer

The following functions are exported:

- `connect_pull_supplier(Object, PullSupplier) -> Return`
  [page 20] Connects pull supplier to the proxy pull consumer
- `disconnect_pull_consumer(Object) -> Return`
  [page 20] Disconnects the ProxyPullConsumer object from the event channel.

CosEventChannelAdmin_ProxyPullSupplier

The following functions are exported:

- `connect_pull_consumer(Object, PullConsumer) -> Return`
  [page 21] Connects pull consumer to the proxy pull supplier
- `disconnect_pull_supplier(Object) -> Return`
  [page 21] Disconnects the ProxyPullSupplier object from the event channel.
- `pull(Object) -> Return`
  [page 21] Transmits data from suppliers to consumers.
- `try_pull(Object) -> Return`
  [page 22] Transmits data from suppliers to consumers.

CosEventChannelAdmin_ProxyPushConsumer

The following functions are exported:

- `connect_push_supplier(Object, PushSupplier) -> Return`
  [page 23] Connects push supplier to the proxy push consumer
- `disconnect_push_consumer(Object) -> Return`
  [page 23] Disconnects the ProxyPushConsumer object from the event channel.
- `push(Object, Data) -> Return`
  [page 23] Communicates event data to the consumers.

CosEventChannelAdmin_ProxyPushSupplier

The following functions are exported:

- `connect_push_consumer(Object, PushConsumer) -> Return`
  [page 25] Connects push consumer to the proxy push supplier
- `disconnect_push_supplier(Object) -> Return`
  [page 25] Disconnects the ProxyPushSupplier object from the event channel.

CosEventChannelAdmin_SupplierAdmin

The following functions are exported:

- `obtain_push_consumer(Object) -> Return`
  [page 26] Creates a ProxyPushConsumer object
- `obtain_pull_consumer(Object) -> Return`
  [page 26] Creates a ProxyPullConsumer object

OrberEventChannel

No functions are exported

OrberEventChannel_EventChannelFactory

The following functions are exported:

- `create_event_channel(Object) -> Return`
  [page 28] Creates a event channel object

CosEventChannelAdmin (Module)

The event service defines two roles for objects: the supplier role and the consumer role. Suppliers supply event data to the event channel and consumers receive event data from the channel. Suppliers do not need to know the identity of the consumers, and vice versa. Consumers and suppliers are connected to the event channel via proxies, which are managed by ConsumerAdmin and SupplierAdmin objects.

There are four general models of communication. These are:

- The canonical push model. It allows the suppliers of events to initiate the transfer of event data to consumers. Event channels play the role of `Notifier`. Active suppliers use event channel to push data to passive consumers registered with the event channel.
- The canonical pull model. It allows consumers to request events from suppliers. Event channels play the role of `Procure` since they procure events on behalf of consumers. Active consumers can explicitly pull data from passive suppliers via the event channels.
- The hybrid push/pull model. It allows consumers request events queued at a channel by suppliers. Event channels play the role of `Queue`. Active consumers explicitly pull data deposited by active suppliers via the event channels.
- The hybrid pull/push model. It allows the channel to pull events from suppliers and push them to consumers. Event channels play the role of `Intelligent agent`. Active event channels can pull data from passive suppliers to push it to passive consumers.

To get access to the record definitions for the structuress use:
`-include_lib("cosEvent/src/CosEventChannelAdmin.hrl")..`

There are seven different interfaces supported in the service:

- ProxyPushConsumer
- ProxyPullSupplier
- ProxyPullConsumer
- ProxyPushSupplier
- ConsumerAdmin
- SupplierAdmin
- EventChannel

IDL specification for CosEventChannelAdmin:

```
#ifndef _COSEVENTCHANELADMIN_IDL
#define _COSEVENTCHANELADMIN_IDL

#include "CosEventComm.idl"

#pragma prefix "omg.org"

module CosEventChannelAdmin
{
  exception AlreadyConnected{};
  exception TypeError{};

  interface ProxyPushConsumer: CosEventComm::PushConsumer
  {
    void connect_push_supplier(in CosEventComm::
                               PushSupplier push_supplier)
                               raises (AlreadyConnected);
  };

  interface ProxyPullSupplier: CosEventComm::PullSupplier
  {
    void connect_pull_consumer(in CosEventComm::
                               PullConsumer pull_consumer)
                               raises (AlreadyConnected);
  };

  interface ProxyPullConsumer: CosEventComm::PullConsumer
  {
    void connect_pull_supplier(in CosEventComm::
                               PullSupplier pull_supplier)
                               raises (AlreadyConnected, TypeError);
  };

  interface ProxyPushSupplier: CosEventComm::PushSupplier
  {
    void connect_push_consumer(in CosEventComm::
                               PushConsumer push_consumer)
                               raises (AlreadyConnected, TypeError);
  };

  interface ConsumerAdmin
  {
    ProxyPushSupplier obtain_push_supplier();
    ProxyPullSupplier obtain_pull_supplier();
  };

  interface SupplierAdmin
  {
    ProxyPushConsumer obtain_push_consumer();
    ProxyPullConsumer obtain_pull_consumer();
  };

  interface EventChannel
```

```
  {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
    void destroy();
  };

};

#endif
```

CosEventChannelAdmin_ConsumerAdmi (Module)

The ConsumerAdmin interface defines the first step for connecting consumers to the event channel. It acts as a factory for creating proxy suppliers. Both consumer administration and supplier administration are defined as separate objects so that the creator of the channel can control the addition of suppliers and consumers.

Any object that possesses an object reference that supports the EventChannelFactory interface can perform the following operations:

Exports

`obtain_push_supplier(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a ProxyPushSupplier object reference.

`obtain_pull_supplier(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a ProxyPullSupplier object reference.

CosEventChannelAdmin_EventChannel (Module)

An event channel is an object that allows multiple suppliers to communicate with multiple consumers in a highly decoupled, asynchronous manner. The event channel is built up incrementally. An event channel factory could be used for creating an event channel. This factory could be found in OrberEventChannel_EventChannelFactory module. When an event channel is created no suppliers or consumers are connected to it. Event Channel can implement group communication by serving as a replicator, broadcaster, or multicaster that forward events from one or more suppliers to multiple consumers.

It is up to the user to decide when an event channel is created and how references to the event channel are obtained. By representing the event channel as an object, it has all of the properties that apply to objects. One way to manage an event channel is to register it in a naming context, or export it through an operation on an object.

Any object that possesses an object reference that supports the ProxyPullConsumer interface can perform the following operations:

Exports

`for_consumers(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a ConsumerAdmin object reference. If ConsumerAdmin object does not exist already it creates one.

`for_suppliers(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a SupplierAdmin object reference. If SupplierAdmin object does not exist already it creates one.

`destroy(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

CosEventChannelAdmin_ProxyPullConsum (Module)

The ProxyPullConsumer interface defines the second step for connecting pull suppliers to the event channel. A proxy consumer is similar to a normal consumer, but includes an additional method for connecting a supplier to the proxy consumer.

There are a number of exceptions that can be returned from functions in this interface.

- AlreadyConnected is defined as `-record('AlreadyConnected', {}).`
- CORBA standard BAD_PARAM is defined as `-record('BAD_PARAM', {'OE_ID', minor, completion_status}).`

The first exception is defined in the file `event_service.hrl` and the second one in the file `corba.hrl`.

Any object that possesses an object reference that supports the ProxyPullConsumer interface can perform the following operations:

Exports

`connect_pull_supplier(Object, PullSupplier) -> Return`

Types:

- Object = #objref
- PullSupplier = #objref of PullSupplier type
- Return = void

This operation connects PullSupplier object to the ProxyPullConsumer object. If a nil object reference is passed CORBA standard `BAD_PARAM` exception is raised. If the ProxyPullConsumer is already connected to a PullSupplier, then the `AlreadyConnected exception` is raised.

`disconnect_pull_consumer(Object) -> Return`

Types:

- Object = #objref
- Return = void

This operation disconnects proxy pull consumer from the event channel and sends a notification about the loss of the connection to the pull supplier attached to it.

CosEventChannelAdmin_ProxyPullSuppl (Module)

The ProxyPullSupplier interface defines the second step for connecting pull consumers to the event channel. A proxy supplier is similar to a normal supplier, but includes an additional method for connecting a consumer to the proxy supplier.

There are a number of exceptions that can be returned from functions in this interface.

- AlreadyConnected is defined as `-record('AlreadyConnected', {}).`
- Disconnected is defined as `-record('Disconnected', {}).`

These exceptions are defined in the file `event_service.hrl`.

Any object that possesses an object reference that supports the ProxyPullSupplier interface can perform the following operations:

Exports

`connect_pull_consumer(Object, PullConsumer) -> Return`

Types:

- Object = #objref
- PullConsumer = #objref of PullConsumer type
- Return = void

This operation connects PullConsumer object to the ProxyPullSupplier object. A nil object reference can be passed to this operation. If so a channel cannot invoke the disconnect_pull_consumer operation on the consumer; the consumer may be disconnected from the channel without being informed. If the ProxyPullSupplier is already connected to a PullConsumer, then the `AlreadyConnected exception` is raised.

`disconnect_pull_supplier(Object) -> Return`

Types:

- Object = #objref
- Return = void

This operation disconnects proxy pull supplier from the event channel. It sends a notification about the loss of the connection to the pull consumer attached to it, unless nil object reference was passed at the connection time.

`pull(Object) -> Return`

Types:

- Object = #objref
- Return = any

This operation blocks until the event data is available or the `Disconnected` exception is raised. It returns the event data to the consumer.

`try_pull(Object) -> Return`

Types:

- Object = #objref
- Return = {any, bool()}

This operation does not block: if the event data is available, it returns the event data and sets the data availability flag to true; otherwise it returns a long with a value of 0 and sets the data availability to false. If the event communication has already been disconnected, the `Disconnected` exception is raised.

CosEventChannelAdmin_ProxyPushCons (Module)

The ProxyPushConsumer interface defines the second step for connecting push suppliers to the event channel. A proxy consumer is similar to a normal consumer, but includes an additional method for connecting a supplier to the proxy consumer.

There are a number of exceptions that can be returned from functions in this interface.

- AlreadyConnected is defined as `-record('AlreadyConnected', {}).`
- Disconnected is defined as `-record('Disconnected', {}).`

These exceptions are defined in the file `event_service.hrl`.

Any object that possesses an object reference that supports the ProxyPushConsumer interface can perform the following operations:

Exports

`connect_push_supplier(Object, PushSupplier) -> Return`

Types:

- Object = #objref
- PushSupplier = #objref of PushSupplier type
- Return = void

This operation connects PushSupplier object to the ProxyPushConsumer object. A nil object reference can be passed to this operation. If so a channel cannot invoke the disconnect_push_supplier operation on the supplier; the supplier may be disconnected from the channel without being informed. If the ProxyPushConsumer is already connected to a PushSupplier, then the `AlreadyConnected exception` is raised.

`disconnect_push_consumer(Object) -> Return`

Types:

- Object = #objref
- Return = void

This operation disconnects proxy push consumer from the event channel. Sends a notification about the loss of the connection to the push supplier attached to it, unless nil object reference was passed at the connection time.

`push(Object, Data) -> Return`

Types:

- Object = #objref
- Data = any
- Return = void

This operation sends event data to all connected consumers via the event channel. If the event communication has already been disconnected, the `Disconnected` is raised.

CosEventChannelAdmin_ProxyPushSupp (Module)

The ProxyPushSupplier interface defines the second step for connecting push consumers to the event channel. A proxy supplier is similar to a normal supplier, but includes an additional method for connecting a consumer to the proxy supplier.

There are a number of exceptions that can be returned from functions in this interface.

- AlreadyConnected is defined as `-record('AlreadyConnected', {}).`
- CORBA standard BAD_PARAM is defined as `-record('BAD_PARAM', {'OE_ID', minor, completion_status}).`

The first exception is defined in the file `event_service.hrl` and the second one in the file `corba.hrl`.

Any object that possesses an object reference that supports the ProxyPushSupplier interface can perform the following operations:

Exports

`connect_push_consumer(Object, PushConsumer) -> Return`

Types:

- Object = #objref
- PushConsumer = #objref of PushConsumer type
- Return = void

This operation connects PushConsumer object to the ProxyPushSupplier object. If a nil object reference is passed CORBA standard `BAD_PARAM` exception is raised. If the ProxyPushSupplier is already connected to a PushConsumer, then the `AlreadyConnected exception` is raised.

`disconnect_push_supplier(Object) -> Return`

Types:

- Object = #objref
- Return = void

This operation disconnects proxy push supplier from the event channel and sends a notification about the loss of the connection to the push consumer attached to it.

CosEventChannelAdmin_SupplierAdmin (Module)

The SupplierAdmin interface defines the first step for connecting suppliers to the event channel. It acts as a factory for creating proxy consumers. Both consumer administration and supplier administration are defined as separate objects so that the creator of the channel can control the addition of suppliers and consumers.

Any object that possesses an object reference that supports the EventChannelFactory interface can perform the following operations:

Exports

`obtain_push_consumer(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a ProxyPushConsumer object reference.

`obtain_pull_consumer(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns a ProxyPullConsumer object reference.

OrberEventChannel (Module)

There is only one interface supported in the service:

- EventChannelFactory

IDL specification for OrberEventChannel:

```
#ifndef _EVENT_CHANNEL_FACTORY_IDL
#define _EVENT_CHANNEL_FACTORY_IDL

#include "OrberEventChannelAdmin.idl"

#pragma prefix "omg.org"

module OrberEventChannel
{
  interface EventChannelFactory
  {
     OrberEventChannelAdmin::EventChannel create_event_channel();
 };
};

#endif
```

OrberEventChannel_EventChannelFactory (Module)

Any object that possesses an object reference that supports the EventChannelFactory interface can perform the following operations:

Exports

`create_event_channel(Object) -> Return`

Types:

- Object = #objref
- Return = #objref

This operation returns an EventChannel object reference.

List of Figures

Chapter 1: CosEvent User's Guide

1.1 Figure 1: Event service Components 4

1.2 Figure 2: Event service Communication Models 5

Index

Modules are typed in *this way*.
Functions are typed in `this way`.

`connect_pull_consumer/2`
    CosEventChannelAdmin_ProxyPullSupplier, 21

`connect_pull_supplier/2`
    CosEventChannelAdmin_ProxyPullConsumer, 20

`connect_push_consumer/2`
    CosEventChannelAdmin_ProxyPushSupplier, 25

`connect_push_supplier/2`
    CosEventChannelAdmin_ProxyPushConsumer, 23

CosEventChannelAdmin_ConsumerAdmin
    `obtain_pull_supplier/1`, 17
    `obtain_push_supplier/1`, 17

CosEventChannelAdmin_EventChannel
    `destroy/1`, 18
    `for_consumers/1`, 18
    `for_suppliers/1`, 18

CosEventChannelAdmin_ProxyPullConsumer
    `connect_pull_supplier/2`, 20
    `disconnect_pull_consumer/1`, 20

CosEventChannelAdmin_ProxyPullSupplier
    `connect_pull_consumer/2`, 21
    `disconnect_pull_supplier/1`, 21
    `pull/1`, 21
    `try_pull/1`, 22

CosEventChannelAdmin_ProxyPushConsumer
    `connect_push_supplier/2`, 23
    `disconnect_push_consumer/1`, 23
    `push/2`, 23

CosEventChannelAdmin_ProxyPushSupplier
    `connect_push_consumer/2`, 25
    `disconnect_push_supplier/1`, 25

CosEventChannelAdmin_SupplierAdmin
    `obtain_pull_consumer/1`, 26
    `obtain_push_consumer/1`, 26

`create_event_channel/1`
    OrberEventChannel_EventChannelFactory, 28

`destroy/1`
    CosEventChannelAdmin_EventChannel, 18

`disconnect_pull_consumer/1`
    CosEventChannelAdmin_ProxyPullConsumer, 20

`disconnect_pull_supplier/1`
    CosEventChannelAdmin_ProxyPullSupplier, 21

`disconnect_push_consumer/1`
    CosEventChannelAdmin_ProxyPushConsumer, 23

`disconnect_push_supplier/1`
    CosEventChannelAdmin_ProxyPushSupplier, 25

`for_consumers/1`
    CosEventChannelAdmin_EventChannel, 18

`for_suppliers/1`
    CosEventChannelAdmin_EventChannel, 18

`obtain_pull_consumer/1`

CosEventChannelAdmin_SupplierAdmin, 26

`obtain_pull_supplier/1`
CosEventChannelAdmin_ConsumerAdmin, 17

`obtain_push_consumer/1`
CosEventChannelAdmin_SupplierAdmin, 26

`obtain_push_supplier/1`
CosEventChannelAdmin_ConsumerAdmin, 17

OrberEventChannel_EventChannelFactory
`create_event_channel/1`, 28

`pull/1`
CosEventChannelAdmin_ProxyPullSupplier, 21

`push/2`
CosEventChannelAdmin_ProxyPushConsumer, 23

`try_pull/1`
CosEventChannelAdmin_ProxyPullSupplier, 22