

cosEvent

Copyright © 1999-2025 Ericsson AB. All Rights Reserved.
cosEvent 2.2.2
July 16, 2025

Copyright © 1999-2025 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

July 16, 2025

1 cosEvent User's Guide

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

1.1 The cosEvent Application

1.1.1 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.

1.1.2 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

1.2.1 Overview

The cosEvent application is a Event Service compliant with the **OMG** 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**.

1.3 Event Service

1.3.1 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.

1.3.2 Event Service Components

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

Figure 3.1: 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.

1.3.3 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 3.2: 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.

1.3.4 A Tutorial on How to Create a Simple Service

To be able to use the cosEvent application supplier and consumer objects must be implemented, which must inherit from the appropriate interface defined in the **CosEventComm.idl** specification.

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

```
#ifndef _MYCLIENT_IDL
#define _MYCLIENT_IDL
#include <CosEventComm.idl>

module myClientImpl {

  interface ownInterface:CosEventComm::PushConsumer {

    void ownFunctions(in any NeededArguments)
       raises(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`.

1.3.5 How to Run Everything

Below is a short transcript on how to run cosEvent.

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

%% Install cosEvent in the IFR.
cosEventApp:install(),

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

%% Start the cosEvent application.
cosEventApp:start(),

%% Start a channel using the default configuration
Ch = cosEventApp:start_channel(),
%% ... or use configuration parameters.
Ch = cosEventApp:start_channel([{pull_interval, 10}, {maxEvents, 50}]),

%% Retrieve a SupplierAdmin and a ConsumerAdmin.
AdminSupplier = 'CosEventChannelAdmin_EventChannel':for_suppliers(Ch),
AdminConsumer = 'CosEventChannelAdmin_EventChannel':for_consumers(Ch),

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

%% Create a Push Consumer Proxy, which the Client Push Supplier will push
%% events to.
ProxyPushConsumer =
  'CosEventChannelAdmin_SupplierAdmin':obtain_push_consumer(AdminSupplier),

%% Create a Push Supplier Proxy, which will push events to the registered
%% Push Consumer.
ProxyPushSupplier =
  'CosEventChannelAdmin_ConsumerAdmin':obtain_push_supplier(AdminConsumer),


%% Create application Clients. We can, for example, start the Clients
%% our selves or look them up in the naming service. This is application
%% specific.
Consumer = myClientImpl_ownInterface:oe_create(),
Supplier = ...

%% Connect each Client to the corresponding Proxy.
'CosEventChannelAdmin_ProxyPushConsumer':
   connect_push_supplier(ProxyPushConsumer, Supplier),
'CosEventChannelAdmin_ProxyPushSupplier':
   connect_push_consumer(ProxyPushSupplier, Consumer),
```

The example above, exemplifies a event system, i.e., the **Canonical Push Model**, where the Supplier client in some way generates event and pushes them to the proxy. The push supplier proxies will eventually push the events to each Consumer client.

2 Reference Manual

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

cosEventApp

Erlang module

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

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

Exports

`install() -> Return`

Types:

`Return = ok | {'EXCEPTION', E} | {'EXIT', R}`

This operation installs the cosEvent application.

`uninstall() -> Return`

Types:

`Return = ok | {'EXCEPTION', E} | {'EXIT', R}`

This operation uninstalls the cosEvent application.

`start() -> Return`

Types:

`Return = ok | {error, Reason}`

This operation starts the cosEvent application.

`stop() -> Return`

Types:

`Return = ok | {error, Reason}`

This operation stops the cosEvent application.

`start_channel() -> Channel`

Types:

`Channel = #objref`

This operation creates a new instance of a *Event Channel* using the default settings.

`start_channel(Options) -> Channel`

Types:

`Options = [Option]`

`Option = {pull_interval, Seconds} | {typecheck, Boolean} | {maxEvents, Integer} | {blocking, Boolean}`

`Channel = #objref`

This operation creates a new instance of a *Event Channel*

- `{pull_interval, Seconds}` - determine how often Proxy Pull Consumers will check for new events with the client application. The default value is 20 seconds.
- `{typecheck, Boolean}` - if this option is set to true the proxies will check if the supplied client object is of correct type. The default value is false.
- `{maxEvents, Integer}` - this option determine how many events the `ProxyPullSuppliers` will store before discarding events. If the limit is reached events will be discarded in any order. The default value is 300.
- `{blocking, Boolean}` - this option determine the behavior of the channel when handling events internally. If set to `true` the risk of a single event supplier floods the system is reduced, but the performance may also be reduced. The default value is `true`.

`start_channel_link() -> Channel`

Types:

`Channel = #objref`

This operation creates a new instance of a *Event Channel*, which is linked to the invoking process, using the default settings.

`start_channel_link(Options) -> Channel`

Types:

`Options = [Option]`

`Option = {pull_interval, Seconds} | {typecheck, Boolean} | {maxEvents, Integer} | {blocking, Boolean}`

`Channel = #objref`

This operation creates a new instance of a *Event Channel*, which is linked to the invoking process, with settings defined by the given options. Allowed options are the same as for `cosEventApp:start_channel/1`.

`stop_channel(Channel) -> Reply`

Types:

`Channel = #objref`

`Reply = ok | {'EXCEPTION', E}`

This operation stop the target event channel.

CosEventChannelAdmin

Erlang 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 all definitions, e.g., exceptions, include necessary `hrl` files by using:
`-include_lib("cosEvent/include/*.hrl").`

There are seven different interfaces supported in the service:

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

CosEventChannelAdmin_ConsumerAdmin

Erlang 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.

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

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_SupplierAdmin

Erlang 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.

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

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.

CosEventChannelAdmin_EventChannel

Erlang 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. 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.

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

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_ProxyPullConsumer

Erlang 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.

To get access to all definitions, e.g., exceptions, include necessary `hrl` files by using:
`-include_lib("cosEvent/include/*.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 = ok | {'EXCEPTION', E}`
`E = #'CosEventChannelAdmin_AlreadyConnected'{} |`
`#'CosEventChannelAdmin_TypeError'{}`

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 `CosEventChannelAdmin_AlreadyConnected` exception is raised. Implementations of ProxyPullConsumers may require additional interface functionality; if these requirements are not met the `CosEventChannelAdmin_TypeError` exception will be raised.

`disconnect_pull_consumer(Object) -> Return`

Types:

`Object = #objref`
`Return = ok`

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_ProxyPushConsumer

Erlang 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.

To get access to all definitions, e.g., exceptions, include necessary `hrl` files by using:
`-include_lib("cosEvent/include/*.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 = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}}`

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 `CosEventChannelAdmin_AlreadyConnected` exception is raised.

`disconnect_push_consumer(Object) -> Return`

Types:

`Object = #objref`
`Return = ok`

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 = ok | {'EXCEPTION', #'CosEventComm_Disconnected'{}}`

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

CosEventChannelAdmin_ProxyPullSupplier

Erlang 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.

To get access to all definitions, e.g., exceptions, include necessary `hrl` files by using:
`-include_lib("cosEvent/include/*.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 = ok | {'EXCEPTION', #'CosEventChannelAdmin_AlreadyConnected'{}}`

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 `CosEventChannelAdmin_AlreadyConnected` exception is raised.

`disconnect_pull_supplier(Object) -> Return`

Types:

`Object = #objref`
`Return = ok`

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 `CosEventComm_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 an undefined value and sets the data availability to false. If the event communication has already been disconnected, the `CosEventComm_Disconnected` exception is raised.

CosEventChannelAdmin_ProxyPushSupplier

Erlang 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.

To get access to all definitions, e.g., exceptions, include necessary `hrl` files by using:
`-include_lib("cosEvent/include/*.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 = ok | {'EXCEPTION', E}`
`E = #'CosEventChannelAdmin_AlreadyConnected'{} |`
`#'CosEventChannelAdmin_TypeError'{}`

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 `CosEventChannelAdmin_AlreadyConnected` exception is raised. Implementations of ProxyPushSuppliers may require additional interface functionality; if these requirements are not met the `CosEventChannelAdmin_TypeError` exception will be raised.

`disconnect_push_supplier(Object) -> Return`

Types:

`Object = #objref`
`Return = ok`

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.