Spectra ORB
C Edition

Lightweight Event Service Guide

PRISMTECH

Spectra ORB
C Edition

Lightweight Event Service Guide

Part Number: EORBC-EVNTLWG

Doc Issue 24, 3 June 2013

Copyright Notice

Contents

Table of Contents

Preface

About the Lightweight Event Service Guide vii
Contacts viii

Introduction

Description 3
Benefits 3
Features 3
Limitations 4

Chapter 1 **Concepts** 5
1.1 **Basic Concept** 5
1.2 **Architecture** 6
1.2.1 The Details 8
1.2.1.1 Events 8
1.2.1.2 Event Communication Model 8
1.2.1.3 Event Channel 9
1.2.1.4 Admin Objects 9
1.2.1.5 Proxies 9
1.2.1.6 Queues 10
1.2.1.7 Federation 11

Chapter 2 **Running the Service** 13
2.1 **Embedding the Service** 13
2.1.1 Configuration Structure 13
2.1.1.1 Blocking versus Rejecting Events 15
2.1.2 The EORB_EventService_init() Method 16
2.1.3 Simple Example 16
2.2 **Running from the Command Line** 17
2.2.1 Running on a Fixed Endpoint 18
2.2.2 lweventc Source Code 18

Index 23

Preface

About the Lightweight Event Service Guide

The *Lightweight Event Service Guide* describes the Spectra ORB Lightweight Event Service C Edition product and how it can be used as a minimal, lightweight messaging service for resource-constrained, CORBA-based applications using the Spectra ORB.

The *Lightweight Event Service Guide* is included with the Spectra ORB C Edition *Documentation Set* and is intended to be used with the *IDL*, *Reference*, and *User Guides*, as well as the other documents included with the Spectra ORB C Edition product. Please refer to the *Product Guide* for a complete list of documents.

Intended Audience

The *Lightweight Event Service Guide* is intended to be used by developers who wish to integrate the Spectra ORB Lightweight Event Service C Edition into products which comply with OMG standards for object services. Readers who use this guide should have a good understanding of the relevant programming languages (for example C and IDL) and the underlying technologies (such as CORBA).

Organisation

- the *Introduction* provides a general description of the service along with a list of its main features, potential uses, and benefits
- the *Concepts* section describes the service's concepts and architecture
- the *Running the Service* section describes how to run the service programmatically or from the command line

Conventions

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


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

i
Item contains helpful hint or special information.

WIN
Information applies to Windows (*e.g.* XP, Vista, Windows 7) only.

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

C
C language specific.

C++
C++ language specific.

Java
Java language specific.

Hypertext links are shown as *blue italic underlined.*

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

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

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown as small `Courier` font in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to “example” and kind field to an empty string
newName[0] = new NameComponent (“example”, “”);
```

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

Arial Bold indicates user-related actions, *e.g.* **File > Save** from a menu.

Step 1: One of several steps required to complete a task.

Contacts

PrismTech can be reached at the following contact points for information and technical support.

USA Corporate Headquarters	**European Head Office**
PrismTech Corporation 400 TradeCenter Suite 5900 Woburn, MA 01801 USA	PrismTech Limited PrismTech House 5th Avenue Business Park Gateshead NE11 0NG UK
Tel: +1 781 569 5819	Tel: +44 (0)191 497 9900 Fax: +44 (0)191 497 9901

Web: *http://www.prismtech.com*
Technical questions: *crc@prismtech.com* (Customer Response Center)
Sales enquiries: *sales@prismtech.com*

Introduction

Description

The Spectra ORB Lightweight Event Service C Edition implementation is a subset of the Object Management Group's (OMG) Event Service Specification and addresses the needs of resource-constrained environments, such as embedded systems. The Spectra ORB Lightweight Event Service is backwards compatible with the full Event Service.

The Spectra ORB Lightweight Event Service is compliant with the OMG's Lightweight Event Service Specification and satisfies the requirements of the Software Communications Architecture (SCA).

The OMG's Event Service enables data, referred to as events, to be sent and received as messages between distributed software objects in a decoupled fashion. Objects which send messages through an intermediary, such as the Event Service, are referred to as being decoupled. Objects which send messages directly between each other are referred to as being tightly coupled.

Decoupling objects which send messages to each other provides a variety of advantages and benefits over tightly coupling the objects. Advantages can include greatly improved scalability, improvements to maintainability, greater flexibility, and others (see Benefits below). In general terms, decoupling enables events to be transmitted more efficiently and flexibly than when events are sent directly between objects (tightly coupled).

The Spectra ORB Lightweight Event Service would be typically used in sectors and industries that employ highly constrained environments which prevent the use of standard CORBA services, such as in embedded systems.

Benefits

Some of the benefits of the Spectra ORB Lightweight Event Service include:

- small code footprint size
- high performance
- ease of maintenance when adding or removing objects which supply and consume events
- more efficient use of network bandwidth between objects which supply events (*suppliers*) and objects which use or consume events (*consumers*)
- performance increasingly improves over tight coupling as the number of suppliers and consumers increases

Features

The Spectra ORB Lightweight Event Service provides the features listed below. These features are compliant with the OMG's Event Service Specification, Version 1.1 and the OMG's Lightweight Services submission.

- a lightweight, minimal service for use in applications and environments which require small code size and high performance
- decouples the transmission of events between suppliers and consumers by using an *event channel* and *proxies*[1]
- avoidance of poor performance due to polling by using the *push style* event transmission model for event notification

Limitations

The Spectra ORB Lightweight Event Service does not support:

- the *pull* model of event transmission (i.e. only the *push* model is supported)
- typed events

However, the lightweight-version of the service does provide all of the other features of the full Event Service and is backwards compatible with it.

1. The events transmitted must be formatted as CORBA *Any* types.

CHAPTER

1 Concepts

1.1 Basic Concept

There are many situations when an object needs to receive notification that an event has been generated or produced by another object, such as when an alarm control panel of a security system needs to know if a remote alarm has been activated. The object may also need to know details about the event itself so that it can take appropriate action. Using the security system example, the alarm panel may need to know which alarm was activated, its location, the reason for the alarm (break-in, fire, etc.) in order to provide appropriate information to security officers.

Obviously, the objects producing and using the event need to be connected to each other in some fashion so that communication of the event can occur. A simple solution would be to connect the objects together directly: notification of an event occurrence and information about it being communicated *directly* between the two objects. Importantly, these objects would then be *tightly coupled* to each other: changes effecting the communication of the event by one object will directly affect the other object.

Tight coupling performs well when one object is connected to only one other object. If, however, many objects are connected to many others, especially when the number of objects changes, then maintainability, performance, and scalability become serious issues. For example, each time an event *producer* object (e.g. a new alarm) is added, then all event users, or *consumer*, objects (e.g. the alarm panels in the building, at the security firm, in the police or fire stations) will need to be changed, too. In software terms, code for all consumer objects, i.e. the *consumers*, will need to be altered, re-compiled, tested, etc., whenever supplier objects, i.e. the *suppliers*, are added.

Also, communication between tightly coupled objects is *synchronous*, that is before the supplier can send an event, the consumer must be ready to receive it. If a supplier is connected to several consumers, then it must wait for the slowest consumer to receive (or consume) the event before it can proceed.

Decoupling suppliers and consumers through an intermediary can overcome these issues. If new suppliers or consumers are added to the system, then only the intermediary needs to be altered, not each consumer or supplier, respectively.

Further, the intermediary can provide event buffers, or *queues*, and *multi-threading* capabilities in order to enable *asynchronous* communication: events can be sent and received without waiting for the slowest “member of the pack”.

An intermediary can therefore take over the task of communicating events between suppliers and consumers: it can provide a *service* for them, who become its *clients*.

The Event Service was the first service that the OMG specified for the decoupled, asynchronous communication of events between event producer and consumer client objects. By decoupling the objects, through the use of an *event channel* and *proxies*, the Event Service provided improved maintainability, performance and scalability over systems which rely on tightly coupled objects.

Like the OMG’s Event Service, PrismTech’s Lightweight Event Service (for convenience, referred to as the Event Service throughout this document) provides decoupled, asynchronous communication between supplier and consumer client objects. However, the Lightweight Event Service provides only those features which are needed for a constrained size and performance environment.

1.2 Architecture

The Event Service can be considered from two viewpoints:

1. From the journey that an event takes from supplier to consumer, i.e. its transmission path.
2. How the Event Service components are conceptually connected and created.

1.2.0.0.1 Event Transmission

A supplier generates events.

1. The supplier sends the events to a proxy representing the consumer, the *consumer proxy.*
2. The consumer proxy forwards the events to a *supplier admin object*.
3. Numerous consumer proxies can be connected to a single supplier admin object. The supplier admin object sends all events it receives to the *event channel*.
4. The event channel transmits the events to a *consumer admin object*. The consumer admin object then forwards those events to its individual *supplier proxies*.
5. Each supplier proxy sends its events to their respective event consumers (one proxy per consumer).

Figure 1, Basic Event Service Architecture, shows the service’s basic conceptual components and event transmission paths.

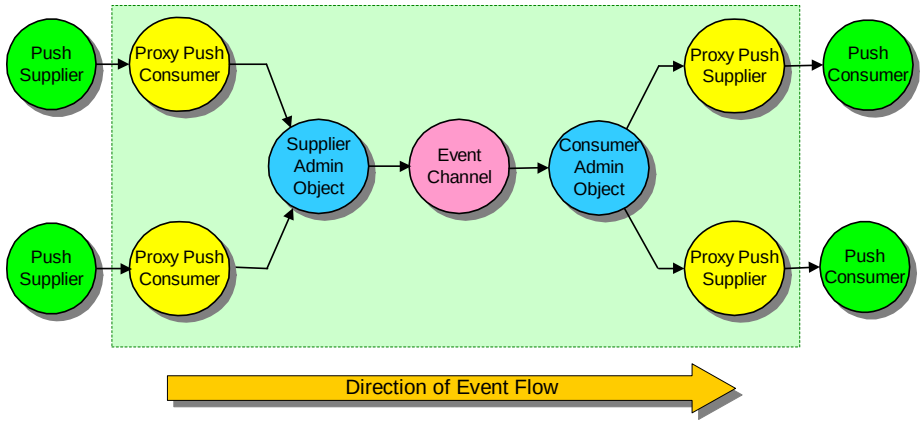

Figure 1 Basic Event Service Architecture

1.2.0.0.2 Component Connection and Creation

The components of the service are organised hierarchically. The main component is the event channel.

Admin objects are created by the event channel; proxies are created from the admin objects. Finally, each proxy is connected to a client supplier object or client consumer object.

1.2.0.0.3 Main Components and Features

The Event Service consists of the following main types of component. Together, they transmit events using a transmission *model*.

The main types of component are event channels, admin objects, proxies, and queues

The type of events which can be transmitted by the Event Service are CORBA *Any*s, which are generally referred to in the context of the Event Service as *untyped* or *Event Style* events.

The transmission model used by the Event Service is the push model (described below).

Figure 2, Main Components, shows the service's main components, including queues.

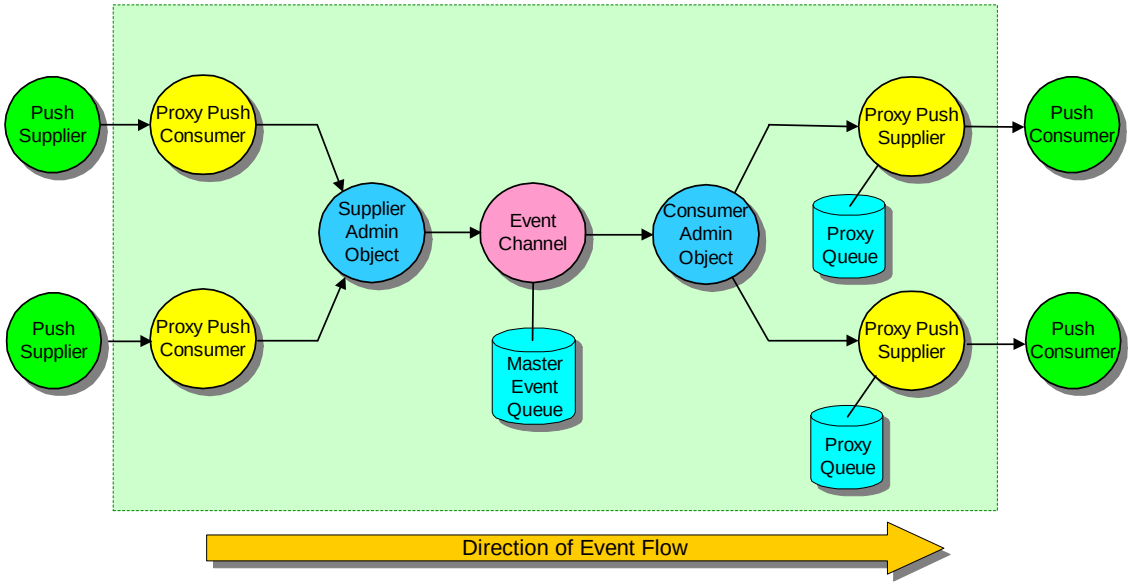

Figure 2 Main Components

These components, event types, transmission models, methods and features will be described in detail below.

1.2.1 The Details

1.2.1.1 Events

The Event Service transmits one type of event: *untyped.*

Untyped events encapsulate basic data types transmitted and received by client objects. Untyped events include the OMG defined *Any* type, which is also referred to as an *Event Style event.*

1.2.1.2 Event Communication Model

The Event Service uses one communication model, the *push model*. In the push model, suppliers actively send or push events to the event channel and consumers passively receive them.

Figure 1, *Basic Event Service Architecture*, on page 7 shows the proxies which are used with the push model, namely the *proxy push consumer* and *proxy push supplier.*

1.2.1.3 Event Channel

The event channel is the component which provides the loosely coupled communication between client objects: it is the event channel which handles supplier registration and broadcasting of events to consumers.

Each Event Service server provides a single event channel.

The event channel creates admin objects, which in turn create proxies. This creation process forms a channel - admin - proxy hierarchy.

1.2.1.4 Admin Objects

Admin objects perform administrative and management functions, such as creating proxies.

Admin objects are associated with either suppliers or consumers (referred to as supplier admin objects or consumer admin objects).

i Note that *supplier* admin objects create *consumer* proxies, and vice versa (remembering that suppliers connect to consumer proxies, consumers connect to supplier proxies).

An event channel in the Event Service has one admin object for suppliers and one admin object for consumers. Each admin object can have any number of proxies.

Admin objects manage or administer the proxies that they have created.

1.2.1.5 Proxies

Proxies connect supplier and consumer client objects to the event channel of the Event Service. Importantly, proxies represent or stand-in for a client. For example, a supplier behaves as if it is connected to an actual consumer, however it is actually connected to a proxy for the consumer, i.e. a *consumer proxy*[1]: suppliers connect to consumer proxies; consumers connect to supplier proxies.

The Event Service uses only the `ProxyPushConsumer` and `ProxyPushSupplier` proxy types.

1.2.1.5.1 Connection and Disconnection States

Figure 3, Proxy States, illustrates the states a proxy can have during creation and disconnection.

1. Also called *proxy consumer*: both forms are used in the OMG specification

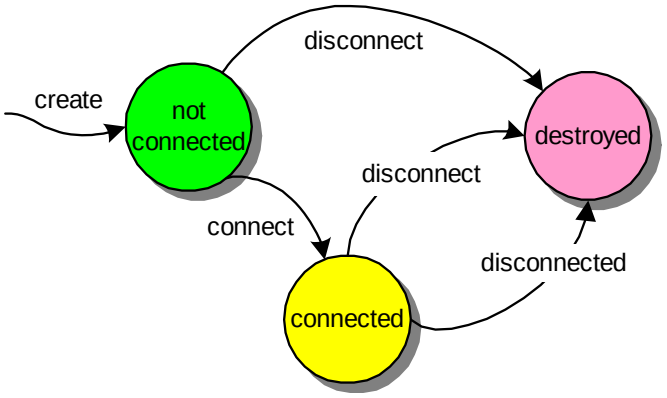

Figure 3 Proxy States

A proxy is a communication end point and disconnecting it implies that the proxy object is destroyed. After being disconnected, the proxy can no longer be used to send or receive events.

A push consumer can also disconnect a proxy by raising the `Disconnected` exception in the push operation.

It is the client's responsibility to disconnect (and therefore destroy) the proxy when the client terminates since the service has no means of knowing that the client no longer exists. Accordingly, the client should call its associated proxy's disconnect method. For example, if the client is a push supplier connected to a *`ProxyPushConsumer`* (suppliers connect to proxy consumers, consumers connect to proxy suppliers), then the *`disconnect_push_consumer()`* method for its `ProxyPushConsumer` object should be called prior to termination.

1.2.1.6 Queues

Queues are buffers for storing events until consumers are ready to receive the events. Queues free suppliers from the need to wait for consumers to consume their events before continuing.

The event channel has a master event queue and each proxy supplier has a proxy queue, one proxy queue per consumer object (see Figure 2, *Main Components*, on page 8).

Incoming events enter the master event queue. Events are then dispatched into proxy queues. The event is removed from the master queue after it has been dispatched to the proxy queue.

1.2.1.7 Federation

Federation is a method of connecting separate Event Service instances and their event channels together (see *Figure 4*, *Event Service Federation*).

Federation effectively creates a composite system partitioned into any number of subsystems. Partitioning an event system into multiple *event subsystems* can have a number of advantages:

- Performance:
  - enabling multiple hosts to be used for utilising increased CPU resources
  - providing fan-out to consumers on the local machine

    Sending events to a channel that in turn forwards them to a number of consumers can result in great performance improvements. As an example, if the consumers are all on the same machine the events can be sent using one network invocation and a series of local invocations.
- Reliability:
  - avoiding single points of failure

    By having multiple event channels it is possible to avoid single points of failure. Although parts of the system may no longer receive events if an event channel fails, this does not necessarily have to affect other consumers.
- Flexibility:
  - makes it easy to move event subsystems

Figure 4 shows that a supplier proxy can act as a supplier and a proxy consumer can act as consumer. This allows channels to be federated without using special clients that forward events from one channel to another. A proxy supplier can be connected directly to a proxy consumer.

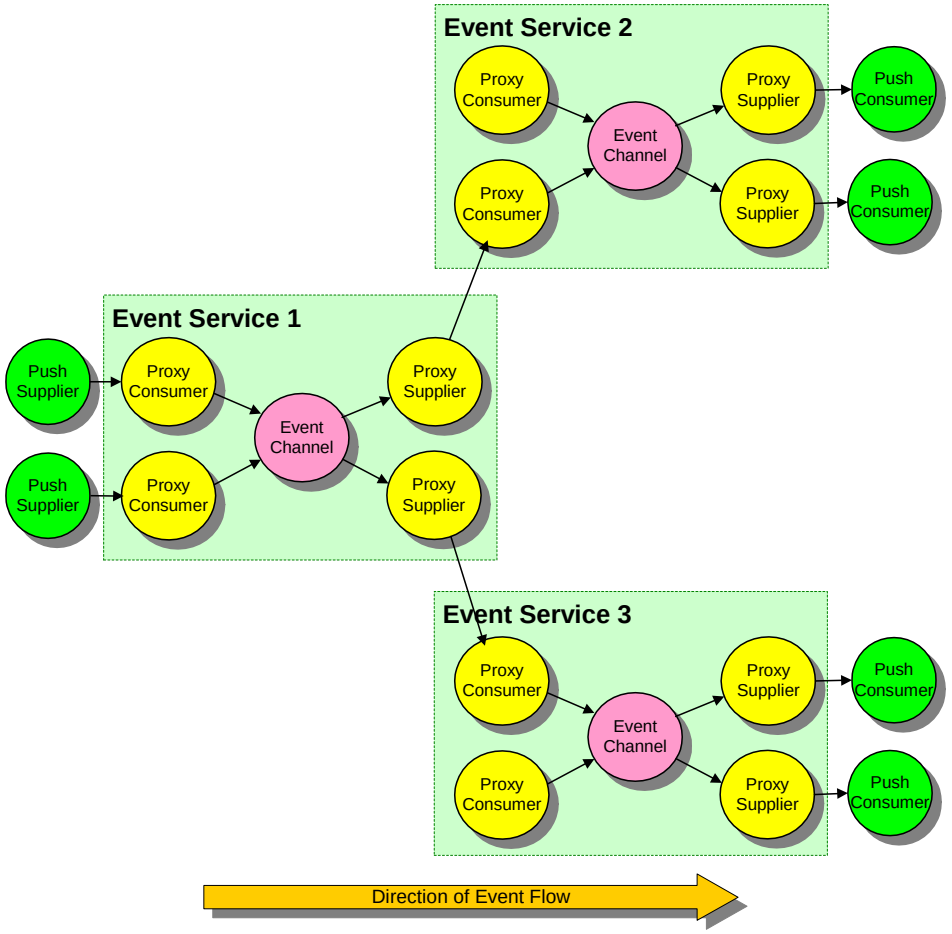

Figure 4 Event Service Federation

CHAPTER

2 Running the Service

The Spectra ORB Lightweight Event Service is usually run by embedding it into an executable or module, however it can also be run from the command line. This section describes how to run the Spectra ORB Lightweight Event Service programmatically by embedding it, plus provides an example of how it can be run from the command line - complete with source code.

2.1 Embedding the Service

The following basic tasks must be performed in order to embed an Event Service instance into an executable or code module:

Step 1: Include the following *include* statement in your code:

```
#include "eOrbC/EORB/EventService.h"
```

Step 2: Ensure your build system has `$(EORBHOME)/include/eOrbC/services/lw` on its include path.

Step 3: Configure your event service instance by setting the property fields in the event service's configuration structure: these properties are used to determine specific aspects of your `EventChannel` instance's behaviour.

Step 4: Instantiate an EventChannel instance: this is achieved by calling the Event Service's `EORB_EventService_init()` method.

2.1.1 Configuration Structure

The structure mentioned in *Step 3:* above defines the property fields described below under Table 1, *Configuration Property Descriptions*. An example of setting the configuration structure fields is shown in *Example 2* on page 15.

if the configuration structure is initialised when it is declared, as shown in *Example 1* and *Example 2* on page 15, then its properties must be initialised in the order that they are shown in *Table 1*. However, if the struct's members are initialised directly by name, as shown in *Example 3* on page 16, then the order that they are initialised does not matter.

Table 1 Configuration Property Descriptions

Property	Description
qosChannelMaxSize	Sets the maximum number of events which will be queued (buffered) by each *EventChannel*. The default value for all platforms, except for VxWorks, is 10240 events. The default value for VxWorks is 100 (due to constraints related to this specific platform).
qosProxyMaxSize	Sets the maximum number of events which will be queued by each *ProxyPushSupplier*. The default value, except for VxWorks, is 1024 events. The default value for VxWorks is 100 (due to constraints related to this specific platform).
qosChannelPushMode	Sets the *EventChannel* push mode to *BLOCK* or *REJECT*. The push mode determines whether push callers will be blocked (*BLOCK*) or their events rejected (*REJECT*) when the channel's queue is full. BLOCK is the default. (See *Blocking versus Rejecting Events* below.)
qosProxyPushMode	Sets the *ProxyPushSupplier*'s push mode to *BLOCK* or *REJECT*. The push mode determines whether the event channel will be blocked from pushing events (*BLOCK*) or the events rejected (*REJECT*) when the proxy's queue is full. BLOCK is the default. (See *Blocking versus Rejecting Events* below.)
qosProxyStackSize	Sets the proxy queue thread's stack size in bytes. The default value of zero (0) sets the stack size to the platform's default stack size.

Example 1

Setting the Configuration Structure Fields' Default Values at Initialisation

```
EORB_EventService_Config config =
{
   EORB_EventService_DEFAULT_CHANNEL_MAX_SIZE,
   EORB_EventService_DEFAULT_PROXY_MAX_SIZE,
   EORB_EventService_BLOCK,
   EORB_EventService_BLOCK,
   0
};
```

Example 2

Setting the Configuration Structure Fields' Other Values at Initialisation

```
EORB_EventService_Config config =
{
   1000,                        /* qosChannelMaxSize */
   100,                         /* qosProxyMaxSize */
   EORB_EventService_REJECT,    /* qosChannelPushMode */
   EORB_EventService_BLOCK,     /* qosProxyPushMode */
   4096                         /* qosProxyStackSize */
};
```

Example 3

Setting the Configuration Structure Field values directly

```
EORB_EventService_Config config;

config.qosChannelMaxSize  = 1000;
config.qosProxyMaxSize    = 100;
config.qosChannelPushMode = EventService.REJECT;
config.qosProxyPushMode   = EventService.BLOCK;
config.qosProxyStackSize  = 4096;
```

2.1.1.1 Blocking versus Rejecting Events

The *block* and *reject* push modes enable users to alter the behaviour of the Event Service to best suit their particular needs regarding how the transmission of events are treated when the event channel and proxy queues are full. Generally, `BLOCK` ensures that all events are received (subject to the limitations of Best Effort Quality of Service[1]) at the expense of the transmission rate being limited to the slowest consumer, whereas `REJECT` ensures the fastest possible speed at the expense of losing events sent to slow consumers.

2.1.1.1.1 Event Channel

If the event channel's queue is full and its push mode is set to *block*, then the clients which are pushing events onto the channel will be blocked (i.e. they must wait until the queue has space to accept events). The block mode ensures that no events are lost.

If the event channel's queue is full and its push mode is set to *reject*, then clients can continue to push events onto the channel. However, events will be rejected (i.e. lost) and a `IMP_LIMIT` exception will be returned to the client for each rejected event. The result is that events will be lost, plus there is the added overhead of returning the exception.

1. *Best Effort* does not guarantee delivery of events in accordance with the OMG Event Service Specification. Refer to the OMG Event Service Specification for details.

2.1.1.1.2 Proxy Push Supplier

If the *ProxyPushSupplier*'s queue is full and its push mode is set to *block*, then the event channel will be blocked from sending events to *any* proxy. The block mode ensures that no events are lost (subject to Best Effort), however the rate that events can be sent to consumers is limited to the rate of the slowest consumer.

If a *ProxyPushSupplier*'s queue is full and its push mode is set to *reject*, then the event channel will continue to push events to all consumer clients connected to the channel, however events will be rejected (i.e. lost) by the `ProxyPushSupplier` whose queue is full. Consumers which can accept events more quickly are allowed to continue to receive events, but the (slower) consumer will lose events.

2.1.2 The EORB_EventService_init() Method

The Event Service's `EORB_EventService_init()` method initialises an Event Service instance. The method takes two parameters:

- a reference to the orb object the service is to be run on
- a reference to the poa which the service will run in
- the configuration structure (described under *Configuration Structure* on page 13)

The method returns the instance's reference. This reference can be used directly, or be stored or pushed for use by other methods, modules, etc.

Example 4

Using the `EORB_EventService_init()` Method

The following code extract creates an Event Service (channel) instance using the `EORB_EventService_init()` method. The `EORB_EventService_init()` method takes an ORB object (created using `CORBA_ORB_init()`), a POA object or NULL and a configuration structure instance (see *Example 2* on page 15). If the POA is NULL then the service will resolve the RootPOA from the ORB.

```
CosEventChannelAdmin_EventChannel channel;

channel = EORB_EventService_init (orb, poa, &config, &env);
```

2.1.3 Simple Example

The source code for a simple example showing how the Event Service can be run from the command line is provided under *lweventc Source Code* on page 18.

Complete source code examples which illustrate the use of the Event Service are also provided in `examples/c/services/event`.

2.2 Running from the Command Line

The Event Service can run from the command line using the *lweventc* example executable with zero or more of the options listed in *Table 2*. These options are the command line equivalent of the embedded service's configurable properties.

```
%  lweventc [options]
```

The complete source code for lweventc is shown following *Table 2*.

Blocking versus Rejecting Events, below, should be read before using the *-EventServiceChannelPushMode* or *-EventServiceChannelPushMode* options.

Table 2 Command Line Options

Option	Description
-EventServiceChannelMaxSize <num>	Sets the maximum number of events, <*n*>, which will be queued (buffered) by the *EventChannel*. The default value for all platforms, except for VxWorks, is 10000 events. The default value for VxWorks is 100 (due to constraints related to this specific platform).
-EventServiceChannelPushMode <BLOCK | REJECT>	Sets the *EventChannel* push mode. The push mode determines whether push callers will be blocked (*BLOCK*) or their events rejected (*REJECT*) when the channel's queue is full. BLOCK is the default. The default is BLOCK. (See *Blocking versus Rejecting Events* on page 15.)
-EventServiceProxyMaxSize <num>	Sets the maximum number of events, <*n*>, which will be queued by the *ProxyPushSupplier*. The default value for all platforms, except for VxWorks, is 1024 events. The default value for VxWorks is 100 (due to constraints related to this specific platform).
-EventServiceChannelPushMode <BLOCK | REJECT>	Sets the *ProxyPushSupplier*'s push mode. The push mode determines whether the event channel will be blocked from pushing events (*BLOCK*) or the events rejected (*REJECT*) when the proxy's queue is full. The default is BLOCK. (See *Blocking versus Rejecting Events* on page 15.)

Table 2 Command Line Options

Option	Description
`-EventServiceProxyStackSize <num>`	Sets the *`ProxyPushSupplier`* thread's stack size in bytes. The default value of zero (0) sets the stack size to the platform's default stack size. The default is `0`.
`-EventServiceUIOP`	Runs the service on a UIOP endpoint. Only available on systems where UIOP is a supported transport. The default is `no`.

2.2.1 Running on a Fixed Endpoint

The server can be run on a fixed endpoint by running with the `-ORBPOAEndpoints` argument. The Event Service servant is created within a child POA *EventService* so for example can be run on a fixed IIOP endpoint with:

```
-ORBPOAEndpoints EventService:iiop:<host>:<port>
```

and resolved as an initial reference by a client using:

```
-ORBInitRef EventService=corbaloc:iiop:<host>:<port>/EventService
```

2.2.2 lweventc Source Code

The main source code for the `lweventc` executable is shown below (the code for command line argument parsing has been omitted). The Spectra ORB convenience utility, *`EORB_IOR_write()`*, is used to write the event channel instance's IOR to a file (located in the local directory where `lweventc` is run), enabling the other programs and modules to access the channel.

The `config.qosChannelMaxSize` and `config.qosProxyMaxSize` properties must be set to values less than or equal to 100 on VxWorks.

```
#include "eOrbC/EORB/EventService.h"
#include "eOrbC/PortableServer/POA.h"

EORB_MAIN (event)
{
   CORBA_ORB orb;
   PortableServer_POA poa;
   PortableServer_POAManager poaManager;
   CosEventChannelAdmin_EventChannel channel = CORBA_OBJECT_NIL;
   CORBA_Environment env;

   /* set default config values */

   EORB_EventService_Config config;

   config.qosChannelMaxSize =
EORB_EventService_DEFAULT_CHANNEL_MAX_SIZE;
   config.qosProxyMaxSize =
EORB_EventService_DEFAULT_PROXY_MAX_SIZE;
   config.qosChannelPushMode = EORB_EventService_BLOCK;
   config.qosProxyPushMode = EORB_EventService_BLOCK;
   config.qosProxyStackSize = 0;

   /* Hook in Minimum POA and TCP transport */
   EORB_plugin (EORB_POA);
   EORB_plugin (EORB_TCP);
   EORB_plugin (EORB_IIOP);
   EORB_plugin (EORB_Any);

   orb = CORBA_ORB_init (&argc, argv, "eorb-ce", &env);
   EORB_CHECK_EXC_RETURN_VAL ("CORBA_ORB_init", &env, -1);

   channel = EORB_EventService_init (orb, &config, &env);
   poa = CORBA_ORB_resolve_initial_references (orb, "RootPOA", &env);
   EORB_CHECK_EXC_RETURN_VAL ("EORB_EventService_init", &env, -1);

   CORBA_Object_release (channel, &env);


   poaManager = PortableServer_POA__get_the_POAManager (poa, &env);
   EORB_CHECK_EXC_RETURN_VAL ("POA__get_the_POAManager", &env, -1);

   PortableServer_POAManager_activate (poaManager, &env);
   EORB_CHECK_EXC_RETURN_VAL ("PortableServer_POAManager_activate",
     &env, -1);

   CORBA_ORB_run (orb, &env);
   EORB_CHECK_EXC_RETURN_VAL ("CORBA_ORB_run", &env, -1);
   return 0;
}
```

Index

Index

A

Admin Objects 9
Architecture 6

B

Basic Concept 5
Basic Event Service Architecture 7
Benefits 3

C

Component Connection and Creation 7
Concepts 5
Connection and Disconnection States 9

E

Event Transmission 6
Event Channel 9
Event Communication Model 8
Event Service Federation 12
Event Transmission 6
Events 8

F

Federation 11

M

Main Components 8
Main Components and Features 7

P

Proxies 9
Proxy States 10

Q

Queues 10

R

Running from the Command Line 17
Running the Service 13