OrbixEvents Programmer's Guide

IONA Technologies PLC
September 2000

M 2 4 7 4

Contents

Preface

OrbixEvents implements the Common Object Request Broker Architecture (CORBA) Event Service which is defined as part of the CORBAservices specification. The CORBAservices specification extends the core CORBA specification with a set of services commonly required in Object Request Broker (ORB) applications. OrbixEvents supports the Internet Inter-ORB Protocol (IIOP) for interoperable communications between CORBA implementations. Consequently, any IIOP-compliant ORB may interact with OrbixEvents.

Orbix documentation is periodically updated. New versions between releases are available at this site:

`http://www.iona.com/docs/orbix/orbix33.html`

If you need assistance with Orbix or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to `doc-feedback@iona.com`.

Audience

The *OrbixEvents Programmer's Guide* is intended for use by ORB application programmers who want to take advantage of the application communications model defined by the CORBA Event Service specification. This guide provides a detailed description of the Event Service communications model and describes how OrbixEvents implements this model.

This guide assumes that you are familiar with both the C++ programming language and with CORBA distributed programming. An OrbixEvents installation requires an existing Orbix or OrbixWeb installation but familiarity with either of these ORB implementations is not strictly necessary.

Organization of this Guide

The *OrbixEvents Programmer's Guide* consists of the following chapters and appendices.

Chapter 1, "Introduction to the CORBA Event Service"

This chapter provides an introduction to the concepts of OrbixEvents. In particular, it introduces the communications model defined by the CORBA Event Service specification.

Chapter 2, "The Programming Interface to the Event Service"

The interfaces to the CORBA Event Service are defined in IDL. This chapter describes these interfaces in detail.

Chapter 3, "OrbixEvents"

This chapter provides an overview of how OrbixEvents implements the CORBA Event Service specification.

Chapter 4, "Programming with the Untyped Push Model"

This chapter describes how to develop an OrbixEvents application that uses the Push model to transmit *untyped* events.

Chapter 5, "Programming with the Typed Push Model"

This chapter describes how to develop of an OrbixEvents application that uses the Push model to transfer *typed* events.

Chapter 6, "Programming with the Untyped Pull Model"

This chapter describes how to develop of an OrbixEvents application that uses the Pull model to transfer *untyped* events.

Chapter 7, "Compiling and Running an OrbixEvents Application"

This chapter explains how to compile and run an OrbixEvents application.

Chapter 8, "OrbixEvents Configuration"

The configuration overhead associated with OrbixEvents is minimal. However, Chapter 8, "OrbixEvents Configuration" addresses some issues associated with configuring an OrbixEvents based application.

Appendix A, "Event Service IDL Definitions"

CORBA defines the programming interface to the Event Service in IDL. The IDL definitions associated with the CORBA Event Service are referenced throughout this guide. Appendix A lists these definitions in full.

Appendix B, "Configuration File Settings"

This appendix lists the configuration settings that you can adjust with the Orbix configuration tool.

Appendix C, "OrbixEventsAdmin::ChannelManager"

OrbixEvents extends the CORBA programming interface to allow you to create and manage *event channels* within an OrbixEvents server. Appendix C describes this interface.

Document Conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`your_name`*

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, no prompt is used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, or Windows 95 command prompt.
... . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.

[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

1

Introduction to the CORBA Event Service

The CORBA Event Service specification defines a model of communication that allows an application to send an event that will be received by any number of objects. The model provides two approaches to initiating event communication. For each of these approaches, event communication can take two forms. This chapter introduces the terminology and concepts that are used throughout this guide.

OrbixEvents implements the CORBA Event Service specification. This specification defines a model for communications between ORB applications that supplements the direct operation call system that client/server applications normally use.

This chapter introduces the basic concepts of the CORBA Event Service communications model. Later chapters will describe the programming interface in detail and show how to implement applications that use the CORBA Event Service using OrbixEvents.

Communications using the CORBA Event Service

Figure 1.1 illustrates the standard CORBA model for communication between distributed applications.

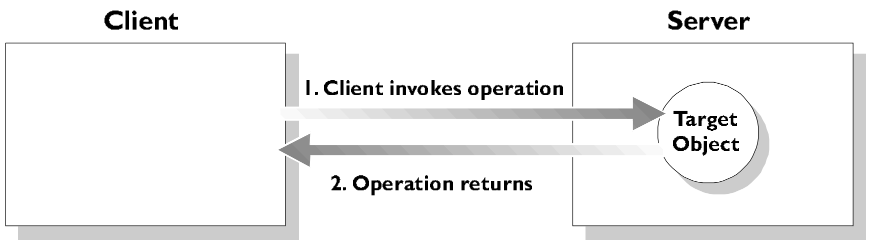

Figure 1.1: *CORBA Model for Basic Client/Server Communications*

In this model, a client application calls an IDL operation on a specified object in a server. The client waits for the call to complete and then receives confirmation of the return status. For any operation call there is a single client and a single server, and each must be available for the call to succeed.

This simple, one-to-one communication model is fundamental to the CORBA architecture. However, some ORB applications need a more complex, indirect communication style. The CORBA Event Service defines a communication model that allows an application to send a message to objects in other applications without any knowledge about the objects that receive the message.

The CORBA Event Service introduces the concept of *events* to CORBA communications. An event originates at an event *supplier* and is transferred to any number of event *consumers*. Suppliers and consumers are completely decoupled: a supplier has no knowledge of the number of consumers or their identities, and consumers have no knowledge of which supplier generated a given event.

In order to support this model, the CORBA Event Service introduces to CORBA a new architectural element, called an *event channel.* An event channel mediates the transfer of events between the suppliers and consumers as follows:

1. The event channel allows consumers to register interest in events, and stores this registration information.
2. The channel accepts incoming events from suppliers.
3. The channel forwards supplier-generated events to registered consumers.

Suppliers and consumers connect to the event channel and not directly to each other (Figure 1.2). From a supplier's perspective, the event channel appears as a single consumer; from a consumer's perspective, the event channel appears as a single supplier. In this way, the event channel decouples suppliers and consumers.

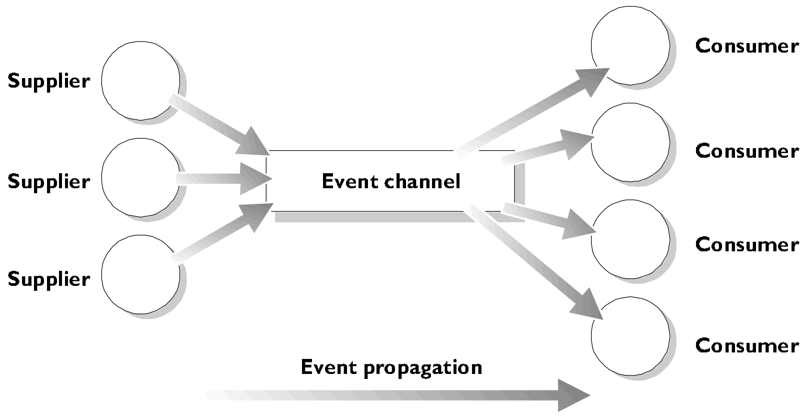

Figure 1.2: *Suppliers and Consumers Communicating through an Event Channel*

Any number of suppliers can issue events to any number of consumers using a single event channel. There is no correlation between the number of suppliers and the number of consumers, and new suppliers and consumers can be easily added to the system. In addition, any supplier or consumer can connect to more than one event channel.

A typical example that uses an event-based communication model is that of a spreadsheet cell. Many documents may be linked to a spreadsheet cell and these documents need to be notified when the cell value changes. However, the spreadsheet software should not need knowledge of each document linked to the cell. When the cell value changes, the spreadsheet software should be able to issue an event which is automatically forwarded to each connected document.

CORBA defines the Event Service at a level above the ORB architecture. Suppliers, consumers and event channels may be implemented as ORB applications, while events are defined using standard IDL operation calls. Suppliers, consumers and event channels each implement clearly defined IDL interfaces that support the steps required to transfer events in a distributed system.

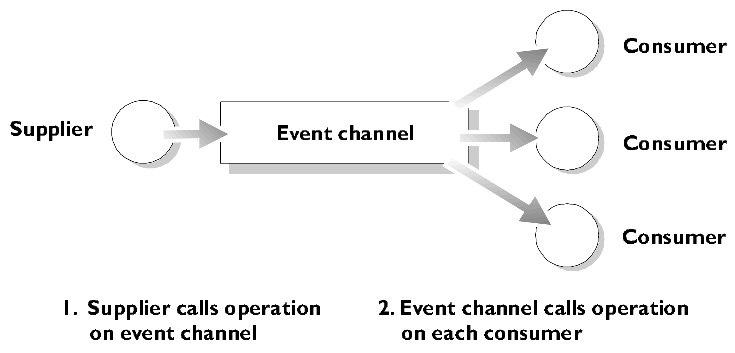
Supplier
Event channel
Consumer
Consumer
Consumer
1. Supplier calls operation on event channel
2. Event channel calls operation on each consumer

Figure 1.3: *An Example Implementation of Event Propagation*

Figure 1.3 illustrates an example implementation of event propagation in a CORBA system. In this example, suppliers are implemented as CORBA clients; the event channel and consumers are implemented as CORBA servers. An event occurs when a supplier invokes a clearly defined IDL operation on an object in the event channel application. The event channel propagates the event by invoking a similar operation on objects in each of the consumer servers. To make this possible, the event channel application stores a reference to each of the consumer objects, for example, in an internal list.

This is not the only way in which the concept of events can map to a CORBA system. In particular, the CORBA Event Service identifies two approaches to initiating the propagation of events, and these affect the implementation architecture. “Initiating Event Communication” on page 5 addresses this topic in detail.

“Types of Event Communication” on page 9 discusses how events can map to IDL operation calls, and describes how you can associate data with an event using IDL operation parameters.

Initiating Event Communication

CORBA specifies two approaches to initiating the transfer of events between suppliers and consumers. These approaches are called the *Push model* and the *Pull model*. In the Push model, suppliers initiate the transfer of events by sending those events to consumers. In the Pull model, consumers initiate the transfer of events by requesting those events from suppliers.

This section illustrates each approach in turn, and then describes how these models can be mixed in a single system.

The Push Model

In the Push model, a supplier generates events and actively passes them to a consumer. In this model, a consumer passively waits for events to arrive. Conceptually, suppliers in the Push model correspond to clients in normal CORBA applications, and consumers correspond to servers.

Figure 1.4 illustrates a Push model architecture in which push suppliers communicate with push consumers through an event channel.

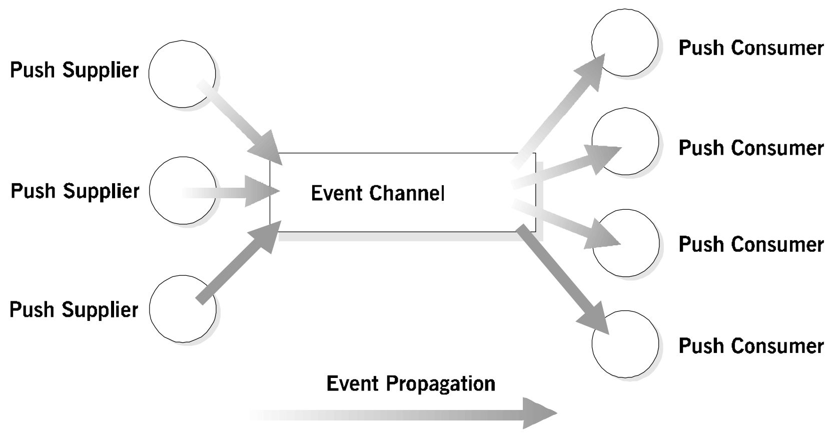

Figure 1.4: *Push Model Suppliers and Consumers Communicating through an Event Channel*

In this architecture, a supplier initiates the transfer of an event by invoking an IDL operation on an object in the event channel. The event channel invokes a similar operation on an object in each consumer that has registered with the channel.

The Pull Model

In the Pull model, a consumer actively requests that a supplier generate an event. In this model, the supplier waits for a pull request to arrive. When a pull request arrives, event data is generated by the supplier and returned to the pulling consumer. Conceptually, consumers in the Pull model correspond to clients in normal CORBA applications and suppliers correspond to servers.

Figure 1.5 illustrates a Pull model architecture in which pull consumers communicate with pull suppliers through an event channel.

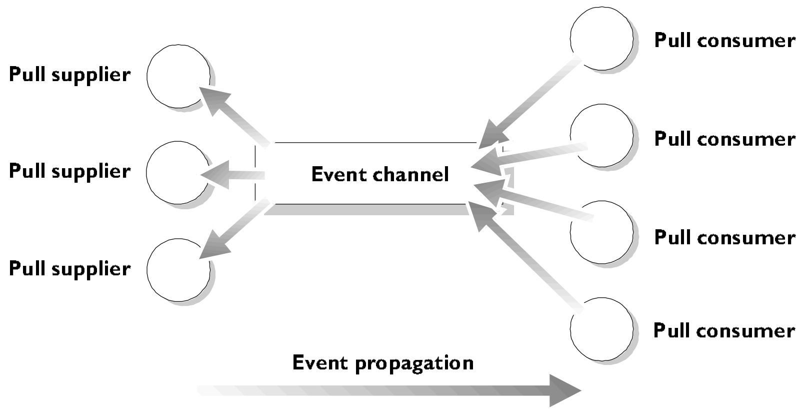

Figure 1.5: *Pull Model Suppliers and Consumers Communicating through an Event Channel*

In this architecture, a consumer initiates the transfer of an event by invoking an IDL operation on an object in the event channel application. The event channel then invokes a similar operation on an object in each supplier. The event data is returned from the supplier to the event channel and then from the channel to the consumer which initiated the transfer.

Mixing the Push and Pull Models in a Single System

Because suppliers and consumers are completely decoupled by an event channel, the Push and Pull models can be mixed in a single system. For example, suppliers may connect to an event channel using the Push model, while consumers connect using the Pull model as shown in Figure 1.6.

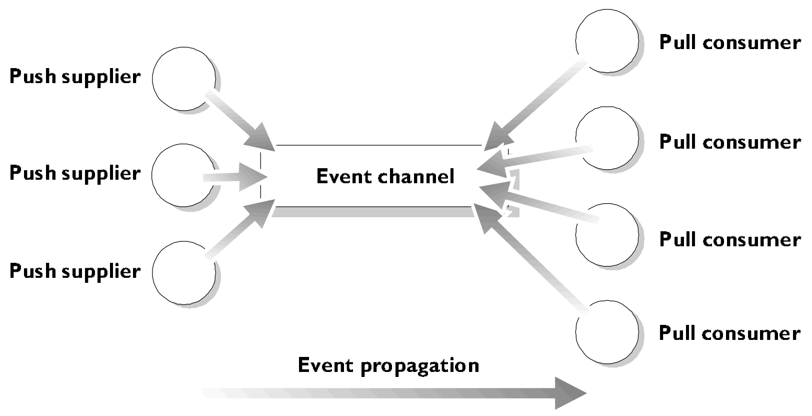

Figure 1.6: *Push Model Suppliers and Pull Model Consumers in a Single System*

In this case, both suppliers and consumers must participate in initiating event transfer. A supplier invokes an operation on an object in the event channel to transfer an event to the channel. A consumer then invokes another operation on an event channel object to transfer the event data from the channel. Unlike the case in which consumers connect using the Push model, the event channel takes no initiative in forwarding the event. The event channel stores events supplied by the push suppliers until some pull consumer requests an event, or until a push consumer connects to the event channel.

Types of Event Communication

The CORBA Event Service maps an event to a successfully completed sequence of operation calls. The operations and the sequence of calls are clearly defined for both Push and Pull models, and data about an event can be passed as operation parameters or return values. This data is specific to each application and is generally not interpreted by implementations of the CORBA Event Service, such as OrbixEvents.

Event communication can take one of the two forms, *typed* or *untyped*.

Untyped Event Communication

In untyped event communication, an event is propagated by a series of generic `push()` or `pull()` operation calls. The `push()` operation takes a single parameter which stores the event data. The event data parameter is of type any, which allows any IDL defined data type to be passed between suppliers and consumers. The `pull()` operation has no parameters but transmits event data in its return value, which is also of type any. Clearly, in both cases, the supplier and consumer applications must agree about the contents of the any parameter and return value if this data is to be useful.

Typed Event Communication

In typed event communication, a programmer defines application-specific IDL interfaces through which events are propagated. Rather than using `push()` and `pull()` operations and transmitting data using an any, a programmer defines an interface that suppliers and consumers use for the purpose of event communication. The operations defined on the interface may contain parameters defined in any suitable IDL data type. In the Push model, event communication is initiated simply by invoking operations defined on this interface. The Pull model is more complex because event communication is initiated by invoking operations on an interface that is specially constructed from the application-specific interface that the programmer defines. Event communication is initiated by invoking operations on the constructed interface.

The form that event communication takes is independent of the method of initiating event transfer. As a consequence, the Push model can be used to transmit typed events or untyped events, and the Pull model can be used to transmit typed or untyped events.

2

The Programming Interface to the Event Service

The CORBA Event Service specification defines a set of interfaces that support the Push and Pull models of initiating the transfer of events in both typed and untyped format. This chapter gives details of these interfaces. The CORBA Event Service specification defines the roles of consumer, supplier and event channel by describing IDL interfaces that each must support. The operations on these interfaces allow consumers and suppliers to register with an event channel to enable the propagation of events.

The CORBA Event Service includes IDL interfaces for both untyped and typed events in both the Push and Pull event models. This chapter describes in detail the IDL interfaces defined for the CORBA Event Service to support these models.

You can find a complete listing of all interfaces relating to the CORBA Event Service in Appendix A, "Event Service IDL Definitions".

The Programming Interface for Untyped Events

The CORBA Event Service for untyped events defines interfaces for suppliers, consumers and event channels. It also defines a number of administration interfaces that allow suppliers and consumers to register with an event channel to allow the transfer of events between them.

Registration of Suppliers and Consumers with an Event Channel

A supplier connects to an event channel to indicate that it wishes to transfer events to consumers through that channel. A consumer connects to an event channel to register its interest in any events supplied through that channel. When a supplier or consumer no longer wishes to send or receive events, the application may disconnect itself from the event channel. In some cases, the event channel may need to disconnect a supplier or consumer explicitly.

The CORBA Event Service defines a set of interfaces that supports untyped event transfer using the Push and Pull models. These interfaces are described in the remainder of this section.

The Push Model for Untyped Events

Four IDL interfaces support connection to and disconnection from event channels using the Push model:

```
PushSupplier
PushConsumer
ProxyPushConsumer
ProxyPushSupplier
```

The interfaces `PushSupplier` and `ProxyPushConsumer` allow suppliers to supply events to an event channel.

The interfaces `PushConsumer` and `ProxyPushSupplier` are specific to consumers, allowing them to receive events from an event channel.

These four interfaces are defined in IDL as follows:

```
// IDL
module CosEventComm {
   exception Disconnected {
   };

   interface PushConsumer {
      void push (in any data) raises (Disconnected);
      void disconnect_push_consumer ();
   };

   interface PushSupplier {
      void disconnect_push_supplier();
   };
};


module CosEventChannelAdmin {
   exception AlreadyConnected {
   };

   exception TypeError {
   };

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

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

   ...
};
```

Connecting a Supplier

A supplier initiates connection to an event channel by obtaining a reference to an object of type ProxyPushConsumer in the channel. The supplier application may wish to be notified if the event channel terminates the connection. If so, the supplier then invokes the operation connect_push_supplier() on that object, passing a reference to an object of type PushSupplier as an operation parameter. If the ProxyPushConsumer is already connected to a PushSupplier, connect_push_supplier() will raise the exception AlreadyConnected.

Connecting a Consumer

A consumer first obtains a reference to a ProxyPushSupplier object implemented in the event channel. In order to register its interest in events from the channel, the consumer then invokes the operation connect_push_consumer() on the ProxyPushSupplier object. The consumer passes a reference to an object of type PushConsumer to the operation call.

If ProxyPushSupplier is already connected to a PushConsumer, connect_push_consumer() will raise the exception AlreadyConnected.

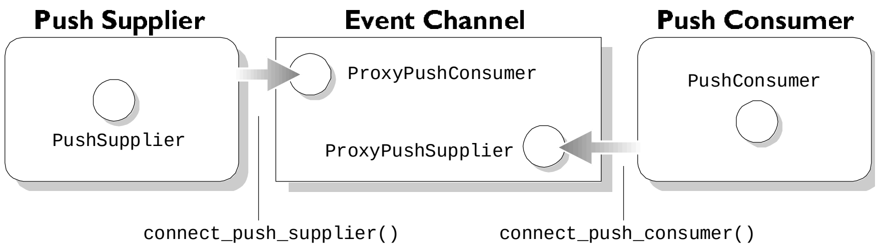

Figure 2.1: *Push Supplier and Push Consumer Connecting to an Event Channel in the Untyped Model*

Figure 2.1 illustrates how a supplier and consumer connect to an event channel. Note that there are no dependencies between the connection of the supplier and the connection of the consumer.

The Pull Model for Untyped Events

A similar set of IDL interfaces supports connection to and disconnection from event channels in the Pull model. These interfaces are:

```
PullSupplier
PullConsumer
ProxyPullConsumer
ProxyPullSupplier
```

The interfaces `PullConsumer` and `ProxyPullSupplier` allow consumers to request events from an event channel.

The interfaces `PullSupplier` and `ProxyPullConsumer` allow an event channel to request events from suppliers.

The Pull model interfaces are defined in IDL as follows:

```
// IDL
module CosEventComm {
   exception Disconnected {
   };

   interface PullSupplier {
      any pull () raises (Disconnected);
      any try_pull (out boolean has_event) raises (Disconnected);
      void disconnect_pull_supplier();
   };

   interface PullConsumer {
      void disconnect_pull_consumer ();
   };
};


module CosEventChannelAdmin {
   exception AlreadyConnected {
   };

   exception TypeError {
   };

   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::PushSupplier pull_supplier)
         raises (AlreadyConnected, TypeError);
   };

   ...
};
```

Connecting a Consumer

In the Pull model, the transfer of events is initiated by consumers. A consumer initiates connection to an event channel by obtaining a reference to an object of type `ProxyPullSupplier` in the channel. The consumer application may wish to be notified if the event channel terminates the connection. If so, it invokes the operation `connect_pull_consumer()` on the `ProxyPullSupplier` object, passing a reference to an object of type `PullConsumer` as an operation parameter. If the `ProxyPullSupplier` is already connected to a `PullConsumer`, `connect_pull_consumer()` raises the exception `AlreadyConnected`.

Connecting a Supplier

To connect to an event channel, a pull supplier first obtains a reference to a `ProxyPullConsumer` object implemented in the event channel. The supplier then invokes the operation `connect_pull_supplier()` on the `ProxyPullConsumer` object, passing a reference to an object of type `PullSupplier` as the operation parameter. If the `ProxyPullConsumer` is already connected to a `PullSupplier`, `connect_pull_supplier()` raises the exception `AlreadyConnected`.

Figure 2.2 illustrates how a pull supplier and pull consumer connect to an event channel. Note that there are no dependencies between the connection of the supplier and the connection of the consumer.

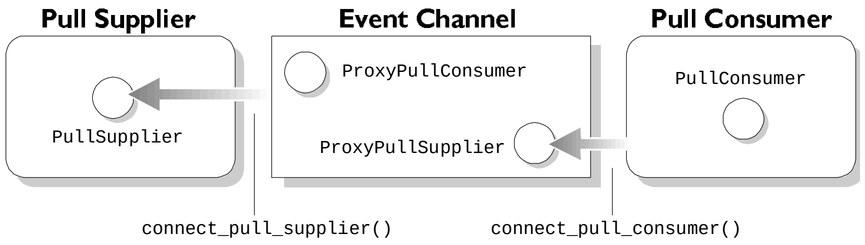

Figure 2.2: *Pull Supplier and Pull Consumer Connecting to an Event Channel in the Untyped Model*

Transfer of Untyped Events Through an Event Channel

The transfer of events from a supplier through an event channel to a consumer follows a simple pattern. Events originate at a supplier. In the Push model, a supplier pushes events into the event channel which in turn pushes the events to registered consumers. In the Pull model, consumers take the active role by requesting events from the event channel; the event channel, in turn, requests events from registered suppliers. Both methods of transfer are described for *untyped* events in the remainder of this section.

The Push Model

The supplier initiates event transfer by invoking the operation `push()` on a `ProxyPushConsumer` object in the event channel, passing the event data as a parameter of type any. The event channel then invokes a `push()` operation on the `PushConsumer` object in each registered consumer, again passing the event data as an operation parameter. Conceptually, this transfer is as shown in Figure 2.3.

Note that the supplier views the event channel as a single consumer and has no knowledge of the actual consumers. Likewise, the consumer views the event channel as a single supplier. In this way, the channel decouples the supplier and consumer.

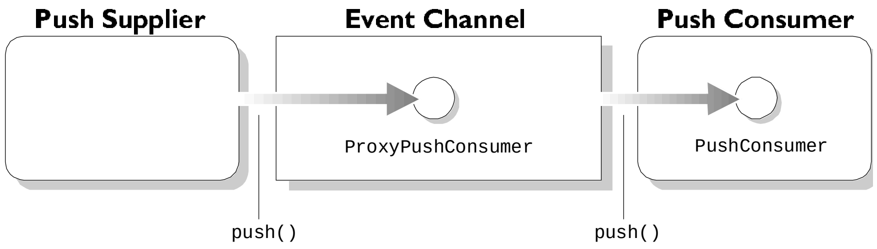

Figure 2.3: *Transfer of an Event Through an Event Channel to a Consumer using the Untyped Push Model*

The Pull Model

The consumer initiates event transfer in the Pull model. The consumer initiates event transfer in one of two ways as described below.

1. `pull()`

   The consumer invokes the `pull()` operation on a `ProxyPullSupplier` object in the event channel.

   The event channel, if it does not already have an event, invokes a `pull()` operation on the `PullSupplier` object in each registered supplier.

   The `pull()` operation blocks until an event is available; the operation then returns the event data in its return value which is of type any. Thus, the consumer application blocks until the event channel can supply an event. The event channel, in turn, blocks until some supplier supplies an event to the channel.

2. `try_pull()`

   The consumer invokes the `try_pull()` operation on a `ProxyPullSupplier` object in the event channel.

   The event channel, in turn, invokes a `try_pull()` operation on the `PullSupplier` object in each registered supplier.

   If no supplier has an event available, `try_pull()` sets its boolean `has_event` parameter to false and returns immediately. If an event is available from some supplier, `try_pull()` sets the `has_event` parameter to true and returns the event data in its return value which is of type any.

Conceptually, the transfer of an event using the Pull model is as shown in Figure 2.4.

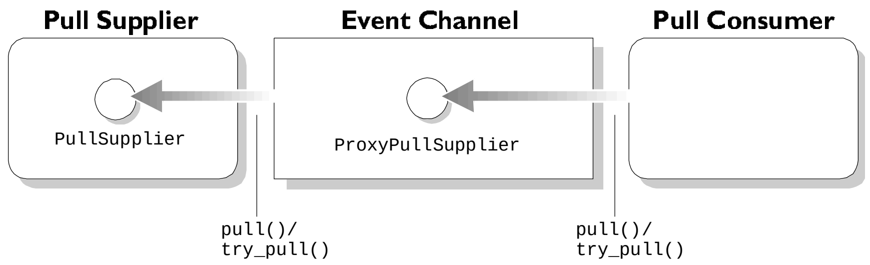

Figure 2.4: *Transfer of an Event Through an Event Channel to a Consumer using the Untyped Pull Model*

Note that, as in the Push model, the channel decouples suppliers and consumers. The consumer views the event channel as a single supplier and has no knowledge of the actual suppliers. Likewise, the supplier views the event channel as a single consumer.

Event Channel Administration Interfaces

The CORBA Event Service specification defines a set of interfaces that support event channel administration. The role of these interfaces is to allow a supplier or consumer to make initial contact with an event channel and to provide a set of standardized operations so that a supplier may obtain a `ProxyPushConsumer` or `ProxyPullConsumer` and a consumer may obtain a `ProxyPushSupplier` or `ProxyPullSupplier` object reference.

Each event channel supports the interface EventChannel, which is defined as follows:

```
// IDL
module CosEventChannelAdmin {
   ...

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

If a supplier or consumer wishes to connect to an event channel, it must first obtain a reference to an EventChannel object in that channel. Typically, the event channel will publish a reference for this object, for example using the CORBA Naming Service.

A supplier then invokes the operation for_suppliers() on the EventChannel object. This operation returns a reference to an object of type SupplierAdmin, which is defined as follows:

```
// IDL
module CosEventChannelAdmin {
   interface SupplierAdmin {
      ProxyPushConsumer obtain_push_consumer ();
      ProxyPullConsumer obtain_pull_consumer ();
   };

   ...
};
```

To obtain a reference to a ProxyPushConsumer object in the event channel, the supplier invokes the operation obtain_push_consumer() on the SupplierAdmin object. At this point, the supplier is ready to connect to the channel and begin transferring events using the Push model.

The supplier invokes the operation obtain_pull_consumer() on the SupplierAdmin object if it wishes to obtain a ProxyPullConsumer. The supplier is then ready to connect to the channel and to transfer events using the Pull model.

Similarly, a consumer invokes the operation `for_consumers()` on an `EventChannel` object in order to obtain a reference to an object of type `ConsumerAdmin`, which is defined as follows:

```
// IDL
module CosEventChannelAdmin {
   interface ConsumerAdmin {
      ProxyPushSupplier obtain_push_supplier ();
      ProxyPullSupplier obtain_pull_supplier ();
   };

   ...
};
```

If the consumer is using the Push model, it then invokes the operation `obtain_push_supplier()` to obtain a reference to a `ProxyPushSupplier`. If the consumer is using the Pull model, it invokes the operation `obtain_pull_supplier()` to obtain a reference to a `ProxyPullSupplier` object in the event channel.

The consumer is then free to register its interest in events propagated through the channel.

The Programming Interface for Typed Events

As described in "Types of Event Communication" on page 9, events can be communicated in untyped form or in typed form. As OrbixEvents supports the Push model for Typed events, this section describes the Push model only.

Using typed event communication, you can define application-specific IDL interfaces through which events can be propagated. You are not restricted to using the operation `push()` to transfer events, and you do not have to pack operation parameters into an IDL any.

The operations you specify in your interfaces may define `in` parameters to allow suppliers to transmit event data. However, since event propagation is uni-directional, these operations may not define `inout` or `out` parameters; they must have a `void` return value and may not have a `raises` clause. These restrictions are the same as the restrictions on oneway operations. However, you do not have to define the operations to be oneway.

The model for typed event communication closely follows the model for untyped events. Typed suppliers connect to a proxy consumer in the event channel and typed consumers connect to a proxy supplier.

Suppliers and consumers must agree on the interface they will use to transfer events. To illustrate this, recall the example of the spreadsheet in "Communications using the CORBA Event Service" on page 2. Many documents can be linked to a spreadsheet cell and these need to be notified of changes to the cell value. The spreadsheet software notifies interested documents of a change to a cell value by generating an event that is forwarded to each connected document. An interface that supports notification of changes to a cell value might be defined as follows:

```
// IDL
interface SpreadsheetCell {
   void value_changed (in float new_value);
   ...
};
```

In this example, documents that are linked to a cell are notified by the spreadsheet software which supplies the event `SpreadsheetCell::value_changed()` whenever the value of a cell changes. The interface `SpreadsheetCell` may define other operations that may be used to supply events to connected documents.

Registration of Suppliers and Consumers with a Typed Event Channel

This section describes how suppliers and consumers register with an event channel in the typed model. The sequence of steps is very similar to that described for the untyped model.

The Typed Push Model

Four IDL interfaces support connection to and disconnection from event channels using the typed Push model:

```
PushSupplier
TypedPushConsumer
ProxyPushSupplier
TypedProxyPushConsumer
```

The interfaces PushSupplier and TypedProxyPushConsumer allow suppliers to supply events to an event channel.

The interfaces TypedPushConsumer and ProxyPushSupplier allow consumers to receive events from an event channel.

PushSupplier and ProxyPushSupplier are as described for the untyped Push model in “Registration of Suppliers and Consumers with an Event Channel” on page 12.

The interfaces TypedPushConsumer and TypedProxyPushConsumer inherit from their counterparts in the untyped Push model. They are defined as follows:

```
// IDL
module CosTypedEventComm {
   interface TypedPushConsumer : CosEventComm::PushConsumer {
      Object get_typed_consumer ();
   };
};

module CosTypedEventChannelAdmin {
   interface TypedProxyPushConsumer :
         CosEventChannelAdmin::ProxyPushConsumer,
         CosTypedEventComm::TypedPushConsumer {
   };
};
```

A typed push supplier initiates connection to an event channel by obtaining a reference to a TypedProxyPushConsumer object in the event channel. The supplier invokes the operation connect_push_supplier() on the TypedProxyPushConsumer object, passing a reference to an object of type PushSupplier as an operation parameter.

A typed push consumer obtains a reference to a `ProxyPushSupplier` object in the event channel and invokes the operation `connect_push_consumer()` on that object, passing an object of type `TypedPushConsumer` as the operation parameter. Figure 2.5 illustrates how a supplier and consumer connect to the event channel.

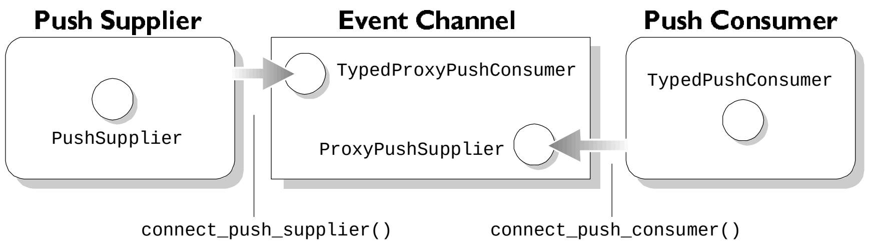

Figure 2.5: *Push Supplier and Push Consumer Connecting to an Event Channel using the Typed Model*

Transfer of Typed Events Through an Event Channel

Once connected to an event channel, suppliers initiate the transfer of typed events in the Push model.

The Typed Push Model

At this point, the typed push supplier is connected to the event channel as described in "Registration of Suppliers and Consumers with a Typed Event Channel" on page 22; specifically, it is connected to a `TypedProxyPushConsumer` object in the event channel.

The `TypedProxyPushConsumer` object is specific to the type of events supplied by the supplier; that is, the supplier and the event channel agree on the type of events supplied by the supplier and accepted by the channel. This agreement is reached when the `TypedProxyPushConsumer` object is set up using the event channel administration interfaces; these interfaces are described in "Typed Event Channel Administration Interfaces" on page 26.

To set up the transfer of events into the channel, the supplier invokes the operation `get_typed_consumer()` on the `TypedProxyPushConsumer` object. The operation `get_typed_consumer()` returns an object reference that supports the interface for which the `TypedProxyPushConsumer` was created. In this example, this is the interface `SpreadsheetCell`. The return type from `get_typed_consumer()` is `CORBA::Object`. Therefore, the supplier must narrow this object reference to obtain a reference of the type for which it supplies events – in this case, `SpreadsheetCell`. Having obtained this object reference, the supplier supplies events to the event channel simply by invoking operations defined in interface `SpreadsheetCell` on the object reference returned by `get_typed_consumer()`. Data associated with the event, if any, is supplied using the operations' `in` parameters. Conceptually, the transfer is as shown in Figure 2.6, where `SpreadsheetCell::value_changed()` events are generated by the supplier.

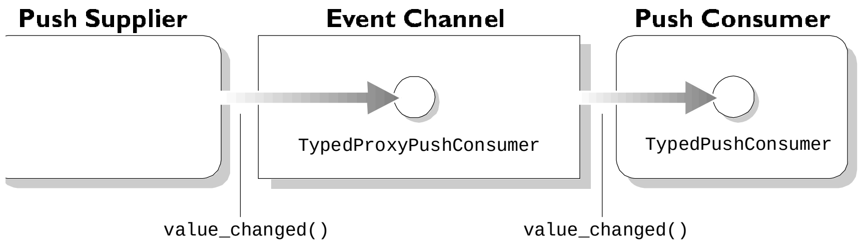

Figure 2.6: *Transfer of an Event Through an Event Channel to a Consumer Using the Typed Push Model*

Typed push suppliers send messages to consumers even though these suppliers do not know anything about the consumers that receive these messages. The flow of information is unidirectional — from suppliers to consumers. Therefore, data associated with an event can be sent by a supplier in an operation's `in` parameters, but no data can be returned because no operation reply can be received by the supplier. Thus, operations invoked by the supplier may not have `inout` or `out` parameters; they must have a `void` return value; and they cannot have a `raises` clause. (These same restrictions apply to `oneway` operations in the standard CORBA model.)

Typed Event Channel Administration Interfaces

To support typed event communication, the CORBA Event Service specification provides a set of administration interfaces similar to those provided for the administration of untyped event channels. Where appropriate, these interfaces use IDL inheritance to indicate that they are specializations of corresponding interfaces in the untyped model.

The interface to a typed event channel is described by the interface `TypedEventChannel`, which is defined as follows:

```
// IDL
module CosTypedEventChannelAdmin {

   interface TypedEventChannel {
      TypedConsumerAdmin for_consumers ();
      TypedSupplierAdmin for_suppliers ();
      void destroy ();
   };

};
```

To connect to a typed event channel, a supplier or consumer must first obtain a reference to a `TypedEventChannel` object in that channel. As for the untyped model, the event channel will typically publish a reference for this object, for example, using the CORBA Naming Service.

A supplier then invokes the operation `for_suppliers()` on the `TypedEventChannel` object. This operation returns a reference to an object of type `TypedSupplierAdmin`, which is defined as follows:

```
// IDL
module CosTypedEventChannelAdmin {

   exception InterfaceNotSupported {
   };

   exception NoSuchImplementation {
   };

   typedef string Key;
   interface TypedSupplierAdmin :
      CosEventChannelAdmin::SupplierAdmin {

      TypedProxyPushConsumer obtain_typed_push_consumer (
               in Key supported_interface)
               raises (InterfaceNotSupported);

      ProxyPullConsumer obtain_typed_pull_consumer (
               in Key uses_interface)
               raises (NoSuchImplementation);
   };

   ...
};
```

The next step is for apush supplier invokes the operation `obtain_typed_push_consumer()` on the `TypedSupplierAdmin` object to obtain a reference to a `TypedProxyPushConsumer` object in the event channel.

Once the supplier has a `TypedSupplierAdmin` object, it is ready to connect to the channel and begin transferring events.

Similarly, a consumer invokes the operation `for_consumers()` on an `TypedEventChannel` object to obtain a reference to an object of type `TypedConsumerAdmin`, which is defined as follows:

```
// IDL
module CosTypedEventChannelAdmin {
   exception InterfaceNotSupported {
   };

   exception NoSuchImplementation {
   };

   typedef string Key;

   interface TypedConsumerAdmin :
            CosEventChannelAdmin::ConsumerAdmin {

      ProxyPushSupplier
      obtain_typed_push_supplier (
               in Key uses_interface)
               raises (NoSuchImplementation);
   };
```

```
        ...
    };
```

The push consumer invokes the operation `obtain_typed_push_supplier()` to obtain a reference to a `ProxyPushSupplier`. The consumer is then free to register its interest in events propagated through the channel.

3

OrbixEvents

OrbixEvents implements the CORBA Event Service specification. This chapter provides an overview of OrbixEvents and its components.

Overview of OrbixEvents

OrbixEvents is implemented as an Orbix server application supporting both untyped and typed events.

To create a CORBA Event Service application using OrbixEvents, you must implement suppliers and consumers using an Object Request Broker (ORB), such as Orbix for C++ or OrbixWeb. These suppliers and consumers communicate through an OrbixEvents server.

An OrbixEvents server can implement one or more conceptual event channels. The criteria that determine the number of event channels required by your application architecture are specific to that application. Some applications may transfer each of several event types through a single channel, while others may have multiple channels that act as alternative sources of a single event type.

Figure 3.1 illustrates an example architecture in which suppliers and consumers communicate through two event channels implemented in a single OrbixEvents server. Note that any given supplier or consumer can connect to multiple event channels simultaneously. In addition, a supplier or consumer can connect to event channels in multiple OrbixEvents servers, if required.

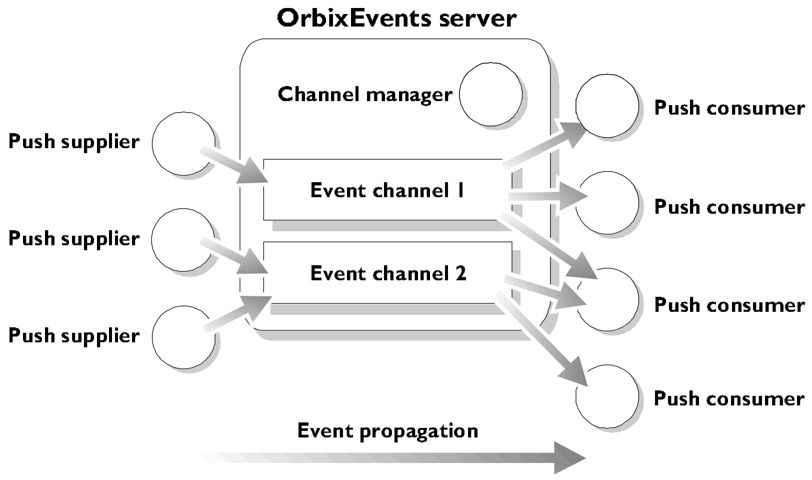

Figure 3.1: *Example OrbixEvents Architecture with Two Event Channels*

An OrbixEvents server maintains an `EventChannel` object, a `SupplierAdmin` object and a `ConsumerAdmin` object for each untyped event channel it implements. (The server maintains the corresponding typed versions, `TypedSupplierAdmin` and `TypedConsumerAdmin`, of these objects for a typed event channel.) An ORB application contacts an event channel by obtaining a reference to the corresponding `EventChannel` object. The application then uses this object to retrieve a reference to the `SupplierAdmin` or the `ConsumerAdmin` object, depending on whether the application is a supplier or consumer.

The `SupplierAdmin` object creates and manages `ProxyPushConsumer` objects for a single untyped event channel. For each supplier that connects to the channel, the `SupplierAdmin` creates a `ProxyPushConsumer` object which the supplier can use to generate events. Similarly, the `ConsumerAdmin` object creates and manages a `ProxyPushSupplier` object for each consumer that connects to the event channel. The typed admin objects create typed versions of these objects in typed event channels.

Each OrbixEvents server also maintains a *channel manager* object. The channel manager supports an interface `OrbixEventsAdmin::ChannelManager` that is specific to OrbixEvents. The `ChannelManager` object allows you to create event channels in the OrbixEvents server. The `ChannelManager` interface is described in full in Appendix C, "OrbixEventsAdmin::ChannelManager".

Components of OrbixEvents

An OrbixEvents consumer or supplier is a normal ORB application that communicates with an OrbixEvents server using standard IDL operation calls.

Consequently, the components of your OrbixEvents implementation include:

- The binary file for the OrbixEvents server.

  The OrbixEvents server is named `es`; it is located in the `bin` directory of your Orbix installation.
- The complete IDL definitions for the CORBA Event Service.

  The IDL definitions for the CORBA Event Service are contained in these files in the `idl` directory:

IDL File	Contents
`cosevents.idl`	This file contains the `CosEventComm` and `CosTypedEventComm` modules.
`coseventsadmin.idl`	This file contains the `CosEventChannelAdmin` and `CosTypedEventChannelAdmin` modules.
`orbixevents.idl`	This file contains the `OrbixEventsAdmin` module.

Table 3.1: *OrbixEvents IDL Files*

4

Programming with the Untyped Push Model

To illustrate the Push model communicating untyped events, this chapter develops a simple application.

As described in Chapter 1, "Introduction to the CORBA Event Service", OrbixEvents allows you to develop Object Request Broker (ORB) applications that communicate using the CORBA Event Service communications model. From a programmer's perspective, the event channel is the key element of a CORBA Event Service application.

This chapter describes an example ORB application that illustrates how you can use OrbixEvents to develop Push model suppliers and consumers that communicate untyped events through event channels.

Overview of an Example Application

The example described in this chapter consists of a push supplier and a push consumer, each of which connects to a single event channel. The supplier repeatedly pushes an event to the event channel and the data associated with each event takes the form of a string. The event channel propagates each event to the consumer, which simply displays the event data. This application is a simple example, but it illustrates a series of development tasks that apply to all OrbixEvents applications.

To develop an OrbixEvents application, you must implement the suppliers and consumers as normal ORB applications that communicate with the event channel through IDL interfaces. OrbixEvents fully implements the event channel, which is created in the OrbixEvents server application. The IDL definitions for the CORBA Event Service are supplied with OrbixEvents.

This chapter describes the implementation of a supplier and consumer using Orbix for C++ as the development ORB. However, the OrbixEvents server fully supports the CORBA Internet Inter-ORB Protocol (IIOP), so you may develop OrbixEvents applications using any IIOP-compatible ORB.

Developing an Untyped Push Supplier

As described in "Transfer of Untyped Events Through an Event Channel" on page 17, a push supplier initiates the transfer of an event by pushing the event into an event channel. The event channel then takes responsibility for forwarding the event to each registered consumer.

This section describes how you can implement a push supplier as an Orbix application that communicates with a single event channel in an OrbixEvents server. This application acts as a client to several IDL interfaces implemented in the OrbixEvents event channel and acts as a server to the interface `PushSupplier`, which it implements.

There are three main programming steps in developing a push supplier:

1. Obtain a reference for a `ProxyPushConsumer` object from the event channel.

   "Obtaining a ProxyPushConsumer from an Event Channel" on page 35 explains this step in detail.
2. Invoke the operation `connect_push_supplier()` on the `ProxyPushConsumer` object, to connect a `PushSupplier` implementation object to the event channel.

   "Connecting a PushSupplier Object to an Event Channel" on page 36 explains this step.
3. Invoke the `push()` operation on the `ProxyPushConsumer` object to initiate the transfer of each event.

   "Pushing Events to an Event Channel" on page 37 explains this step.

"The Push Supplier Application" on page 38 shows how these steps fit into a full Push supplier application.

Obtaining a ProxyPushConsumer from an Event Channel

A push supplier needs to obtain a reference for a ProxyPushConsumer object in an event channel in order to transfer events to the channel for later distribution to consumers. The supplier transfers events by invoking the operation push() on the target ProxyPushConsumer object.

In order to obtain a ProxyPushConsumer object reference from an event channel, a supplier must implement the following programming steps:

1. Obtain a reference to an EventChannel object in the OrbixEvents server.
2. Invoke the operation for_suppliers() on the EventChannel object, in order to obtain a SupplierAdmin object reference.
3. Invoke the operation obtain_push_consumer() on the SupplierAdmin object. This operation returns a ProxyPushConsumer object reference.

In OrbixEvents, every event channel has an associated event channel identifier which can be used to retrieve the channel's EventChannel object reference. When using the Orbix _bind() call, you can specify the channel identifier as the EventChannel object marker value. Chapter 8, "OrbixEvents Configuration" describes in detail how you can associate an identifier with an OrbixEvents event channel. For example:

```
// C++
EventChannel_var channelVar;
char *serverHost;
...

try {
   channelVar = EventChannel::_bind ("Channel_1:ES",
      serverHost);
}
catch (...) {
   // Handle exception.
   ...
}
```

Note that the server name for the OrbixEvents server is ES.

Connecting a PushSupplier Object to an Event Channel

When the supplier has retrieved the EventChannel object reference and used this to obtain a ProxyPushConsumer, the supplier needs to connect an implementation of the PushSupplier interface to the event channel. As described in "Registration of Suppliers and Consumers with an Event Channel" on page 12, this interface is defined as follows:

```
// IDL
module CosEventComm {
   ...

   interface PushSupplier {
      void disconnect_push_supplier ();
   };
};
```

The role of this interface is to allow the event channel to disconnect the supplier by invoking the operation disconnect_push_supplier(). This may happen if the event channel closes down.

In our example, the supplier implements the PushSupplier interface by defining the class PushSupplier_i, for example as follows:

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
...

class PushSupplier_i
   : public virtual CosEventComm::PushSupplierBOAImpl {

   public:
   unsigned char m_disconnected;

   PushSupplier_i () {
      m_disconnected = 0;
   }
   void disconnect_push_supplier (
      CORBA::Environment& env = CORBA::default_environment) {
      m_disconnected = 1;
   }
};
```

This class uses a simple flag mechanism to indicate the connection state of the supplier. The supplier connects an object of this type to an event channel by calling the operation `connect_push_supplier()` on the `ProxyPushConsumer` object.

Pushing Events to an Event Channel

The following code extract from the example supplier program is a simple demonstration of initiating the transfer of events:

```
// C++
while (!psImpl.m_disconnected) {
   CORBA::Any a;
   a <<= eventDataString;
   ppcVar->push (a);
}
```

In this example, the supplier repeatedly pushes an event to the event channel by calling the operation `push()` on a `ProxyPushConsumer` object. The supplier represents the event data using a simple string, but this is not necessary in general. The operation `push()` takes a parameter of type any for the event data, so you may represent this data using any IDL type.

Note that our supplier stops sending events only when it receives an incoming `disconnect_push_supplier()` operation call from the event channel. As an alternative, the supplier could explicitly disconnect from the event channel by invoking the operation `disconnect_push_consumer()` on the event channel `ProxyPushConsumer` object.

The Push Supplier Application

The three main programming steps in the development of push supplier applications have been described in detail.

The following source code illustrates how each of these steps fits in to the full push supplier application.

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include <PushSupplier_i.h>
...

int main(int argc, char** argv) {
   char* eventDataString = "Hello World!";
   CosEventChannelAdmin::EventChannel_var ecVar;
   CosEventChannelAdmin::SupplierAdmin_var saVar;
   CosEventChannelAdmin::ProxyPushConsumer_var ppcVar;
   PushSupplier_i psImpl;
   char *serverHost;

   try {
      //
      // Step 1. Get a ProxyPushConsumer object reference.
      //

      // Obtain an event channel reference.
   try {
      ecVar = EventChannel::_bind ("Channel_1:ES",
         serverHost);
   }
   catch (...) {
      // Handle exception.
   ...
   }
      if (CORBA::is_nil (ecVar))
         return 1;

      // Obtain a supplier administration object.
      saVar = ecVar->for_suppliers ();
      // Obtain a proxy push consumer.
      ppcVar = saVar->obtain_push_consumer ();

      //
      // Step 2. Connect a push supplier implementation object.
      //
      ppcVar->connect_push_supplier (&psImpl);

      //
      // Step 3. Push events to the event channel.
      //
      while (!psImpl.m_disconnected) {
         CORBA::Any a;
         a <<= eventDataString;
         ppcVar->push (a);
         CORBA::Orbix.processNextEvent (1000);
      }

      // When finished, disconnect the consumer.
      ppcVar->disconnect_push_consumer();
   }
   catch (...) {
      // Handle exception
      ...
      return 1;
   }
   return 0;
}
```

Developing an Untyped Push Consumer

A push consumer receives events from an event channel, with no knowledge of the suppliers from which those events originated. An event channel propagates an event to a push consumer by invoking the operation `push()` on a `PushConsumer` implementation object in the consumer application. As such, the main functionality of a push consumer is associated with registering a `PushConsumer` object with an event channel and receiving incoming operation calls on that object.

To develop a push consumer application, you must implement the following steps:

1. Obtain a reference for a `ProxyPushSupplier` object from the event channel.

   "Obtaining a ProxyPushSupplier from an Event Channel" on page 41 explains this step.
2. Connect a `PushConsumer` implementation object to the event channel, by invoking the operation `connect_push_consumer()` on the `ProxyPushSupplier` object.

   "Connecting a PushConsumer Object to an Event Channel" on page 41 explains this step.
3. Monitor incoming operation calls.

   "Monitoring Incoming Operation Calls" on page 44 explains this step.

"The Push Consumer Application" on page 45 shows how these steps fit in to a full Push consumer application.

Obtaining a ProxyPushSupplier from an Event Channel

Each push consumer connected to an event channel receives every event raised by every supplier connected to the channel. However, consumers have no knowledge of the suppliers. Consumers simply connect to an object in the event channel which acts as a single source of events.

This object is responsible for storing a `PushConsumer` object reference for each connected consumer and invoking the `push()` operation on each of these references when a supplier transmits an event. The event channel object which stores consumer references is of type `ProxyPushSupplier`. The first task in developing a push consumer application is to obtain a reference to this object.

There are three stages in obtaining a `ProxyPushSupplier` object reference:

1. Obtain a reference to an `EventChannel` object in the event channel.
2. Invoke the operation `for_consumers()` on the `EventChannel` object to obtain a `ConsumerAdmin` object reference.
3. Invoke the operation `obtain_push_supplier()` on the `ConsumerAdmin` object. This operation returns a `ProxyPushSupplier` object reference.

You may implement the first of these steps in exactly the manner described for push supplier applications in “Obtaining a ProxyPushConsumer from an Event Channel” on page 35. The remaining steps involve normal operation invocations.

Connecting a PushConsumer Object to an Event Channel

When a consumer has obtained a reference to the `ProxyPushSupplier` object in an event channel, the next step is to register a `PushConsumer` implementation object with the `ProxyPushSupplier`. The event channel uses the `PushConsumer` object to propagate events to the consumer.

As described in “Registration of Suppliers and Consumers with an Event Channel” on page 12, the CORBA Event Service specification defines the interface `PushConsumer` as follows:

```
// IDL
module CosEventComm {
   interface PushConsumer {
      oneway void push (in any data) raises (Disconnected);
      void disconnect_push_consumer ();
   };
```

```
    ...
};
```

When an event arrives at an event channel, the channel `ProxyPushSupplier` object invokes the operation `push()` on each connected consumer, passing the event data as an `any` parameter. The `disconnect_push_consumer()` operation allows an event channel to disconnect a consumer, for example if the channel closes down.

Our consumer uses the following example implementation of this interface:

```
// C++
class PushConsumer_i
   : public virtual CosEventComm::PushConsumerBOAImpl {

   public:
   unsigned char m_disconnected;

   PushConsumer_i(){
      m_disconnected = 0;
   }

   virtual void disconnect_push_consumer (
      CORBA::Environment& env = CORBA::default_environment){
      m_disconnected = 1;
   }

   virtual void push (CORBA::Any& any,
      CORBA::Environment& env = CORBA::default_environment){
      char* msg;

      if (a >>= msg)
         cout << "Event received: event data = " << msg << endl;
      else
         cout <<
               "Event received with unexpected event data type."
              << endl;
   }
};
```

This class includes a trivial implementation of the push() operation, through which the consumer receives events. In normal OrbixEvents applications, this operation requires a more complex implementation which reacts appropriately to incoming events. The exact requirements for implementing the push() operation are application specific.

Monitoring Incoming Operation Calls

The main role of the consumer is to receive events from the event channel in the form of IDL operation calls. Consequently, the consumer must monitor and process any incoming calls. The example Orbix consumer application does this by repeatedly calling `processNextEvent()` on the `CORBA::Orbix` object, as follows:

```
// C++
while (!pcImpl.m_disconnected) {
   CORBA::Orbix.processNextEvent ();
}
```

The function `processNextEvent()` handles a single incoming operation call and then returns.

If the consumer receives an invocation on the operation `disconnect_push_consumer()`, then the implementation of this operation sets the value `pcImpl.m_disconnected` to one and breaks the consumer's event processing loop. Consequently, our consumer receives all events until the event channel explicitly forces it to disconnect.

As an alternative, the consumer could explicitly disconnect itself from the event channel when it no longer wishes to receive events. The consumer does this by invoking `disconnect_push_supplier()` on the event channel `ProxyPushSupplier` object.

The Push Consumer Application

The three main programming steps in the development of push consumer applications have been described in detail.

The following source code illustrates how each of these steps fits in to the full push consumer application.

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include <PushConsumer_i.h>

int main(int argc, char** argv) {
   CosEventChannelAdmin::EventChannel_var ecVar;
   CosEventChannelAdmin::ConsumerAdmin_var caVar;
   CosEventChannelAdmin::ProxyPushSupplier_var ppsVar;
   PushConsumer_i pcImpl;
   char *serverHost;

   try {
      //
      // Step 1. Get a ProxyPushSupplier object reference.
      //

      // Obtain an event channel reference.
      try {
      ecVar = EventChannel::_bind ("Channel_1:ES",
         serverHost);
      }
      catch (...) {
         // Handle exception.
   ...
      }
      if (CORBA::is_nil (ecVar))
         return 1;

      // Obtain a consumer administration object.
      caVar = ecVar->for_consumers ();

      // Obtain a proxy push supplier.
      ppsVar = caVar->obtain_push_supplier ();
      //
      // Step 2. Connect a push consumer implementation object.
      //
      ppsVar->connect_push_consumer (&pcImpl);

      //
      // Step 3. Monitor incoming operation calls.
      //
      while (!pcImpl.m_disconnected) {
         CORBA::Orbix.processNextEvent ();
      }

      // When finished, disconnect the supplier.
      ppsVar->disconnect_push_supplier();
   }
   catch (...) {
      // Handle exception.
      ...
      return 1;
   }
   return 0;
}
```

5

Programming with the Typed Push Model

To illustrate the use of the Push model to transmit typed events, this chapter develops a simple example.

This chapter describes how to develop an ORB application using the CORBA Event Service typed communications model that allows programmers to define an application-specific IDL interface. Callers can invoke operations defined on this interface to push events into an event channel. The parameters defined on these operations can specify the IDL data types to be used to pass data on each event so the programmer is not restricted to passing data in an any.

As described in Chapter 2, "The Programming Interface to the Event Service", typed push model suppliers and consumers communicate through event channels.

Overview of an Example Application

Consider a Stock Price application that reports the sales price of stock. The application that reports the sales price is a supplier of events. As well as reporting the price of stock, it may also generate events when the price of a particular stock exceeds a given threshold, when sales activity on the stock rises above a certain level, and so on.

Many different applications might be interested in receiving the events generated by the Stock Price application. These applications are consumers of events. Consumers might include stock brokers, insider trading watchdogs, government departments, and so on.

A suitable interface, supported by consumers of events for a Stock Price application, might be defined as follows:

```
// IDL
interface StockPrice
{
   void quote (in float new_price);
};
```

Using this interface, a supplier application supplies events by invoking the `quote()` operation. The data associated with each event indicates the new price for the stock and takes the form of a `float`.

The simplified example that is described in this chapter consists of a push supplier and a push consumer, each of which connects to a single event channel. The supplier repeatedly pushes `StockPrice::quote()` events to the event channel. The event channel propagates each event to the consumer, which will simply display the event data. This application is simple, but it illustrates a series of development tasks that apply to all OrbixEvents applications using the typed push model.

Because event communication is unidirectional, operations defined on interface `StockPrice` must obey the same restrictions as `oneway` operations, although they do not need to be explicitly declared `oneway`. Thus, the operations' parameters must be `in` parameters; the return value must be `void`; and the operation cannot have a `raises` clause.

When developing an OrbixEvents application, you must implement the suppliers and consumers as normal ORB applications that communicate with the event channel through IDL interfaces. OrbixEvents fully implements the event channel, which is created in the OrbixEvents server application. The IDL definitions for the CORBA Event Service are supplied with OrbixEvents.

This chapter examines the implementation of a supplier and consumer using Orbix for C++ as the development ORB. However, the OrbixEvents server fully supports the CORBA Internet Inter-ORB Protocol (IIOP), so you may develop OrbixEvents applications using any IIOP-compatible ORB.

Developing a Typed Push Supplier

As described in Chapter 2, "The Programming Interface to the Event Service", a push supplier initiates the transfer of an event by pushing the event into an event channel. The event channel then takes responsibility for forwarding the event to each registered consumer.

This section describes how you can implement a typed push supplier as an Orbix application that communicates with a single event channel in an OrbixEvents server. This application acts as a client to several IDL interfaces implemented in the OrbixEvents event channel and acts as a server to the interface `PushSupplier`, which it implements.

There are four main programming steps in developing a typed push supplier:

1. Obtain a reference for a `TypedProxyPushConsumer` object from the event channel.

   "Obtaining a TypedProxyPushConsumer from an Event Channel" on page 50 explains this step in detail.
2. Invoke the operation `connect_push_supplier()` on the `TypedProxyPushConsumer` object, to connect a `PushSupplier` implementation object to the event channel.

   "Connecting a PushSupplier Object to an Event Channel" on page 51 explains this step.
3. Invoke the operation `get_typed_consumer()` on the `TypedProxyPushConsumer` object and narrow the returned `CORBA::Object` object reference to the appropriate application-specific type. "Obtaining a Typed Push Consumer from a ProxyPushConsumer" on page 52 explains this step.
4. Invoke an appropriate operation defined on the application-specific interface on the object reference obtained in Step 3 to initiate the transfer of each event.

   "Pushing Events to an Event Channel" on page 53 explains this step.

Obtaining a TypedProxyPushConsumer from an Event Channel

A typed push supplier needs to obtain a reference for a TypedProxyPushConsumer object in an event channel in order to transfer events to the channel for later distribution to consumers.

To obtain a TypedProxyPushConsumer object reference from an event channel, a supplier must implement the following programming steps:

1. Obtain a reference to a TypedEventChannel object in the event channel.
2. Invoke the operation for_suppliers() on the TypedEventChannel object, in order to obtain a TypedSupplierAdmin object reference.
3. Invoke the operation obtain_typed_push_consumer() on the TypedSupplierAdmin object, passing the name of the interface for which the typed consumer is required as a parameter to the operation. This operation returns a TypedProxyPushConsumer object reference.

These steps are defined in the CORBA Event Service specification and apply to all Event Service implementations.

In OrbixEvents, every event channel has an associated event channel identifier which can be used to retrieve the channel's TypedEventChannel object reference. When using the Orbix _bind() call, you may specify the channel identifier as the TypedEventChannel object marker value. Chapter 8, "OrbixEvents Configuration" describes in detail how you can associate an identifier with an OrbixEvents event channel. For example:

```
// C++
TypedEventChannel_var channelVar;
char *serverHost;
...

try {
   channelVar = TypedEventChannel::
      _bind ("Typed_Channel_1:ES", serverHost);
}
catch (...) {
   // Handle exception.
   ...
}
```

Note that the server name for the OrbixEvents server is ES.

Connecting a PushSupplier Object to an Event Channel

When the supplier has retrieved the TypedEventChannel object reference and used this to obtain a TypedProxyPushConsumer, the supplier needs to connect an implementation of the PushSupplier interface to the event channel. As described in Chapter 2, "The Programming Interface to the Event Service", this interface is defined as follows:

```
// IDL
module CosEventComm {
   ...

   interface PushSupplier {
      void disconnect_push_supplier ();
   };
};
```

The role of this interface is to allow the event channel to disconnect the supplier by invoking the operation disconnect_push_supplier(). This may happen if the event channel closes down.

In our example, the supplier implements the PushSupplier interface by defining the class TypedPushSupplier_i, for example as follows:

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
...

class TypedPushSupplier_i
   : public virtual CosEventComm::PushSupplierBOAImpl {

   public:
   unsigned char m_disconnected;

   TypedPushSupplier_i () {
      m_disconnected = 0;
   }
   void disconnect_push_supplier (
      CORBA::Environment& env = CORBA::default_environment) {
      m_disconnected = 1;
   }
};
```

This class uses a simple flag mechanism to indicate the connection state of the supplier. The supplier connects an object of this type to an event channel by calling the operation `connect_push_supplier()` on the `TypedProxyPushConsumer` object.

Obtaining a Typed Push Consumer from a ProxyPushConsumer

To send typed events, the supplier must obtain a reference to an object in the event channel that supports the `StockPrice` interface. The supplier does this by invoking the operation `get_typed_consumer()` on the `ProxyPushConsumer` object it got from the event channel.

```
// C++
CORBA::Object_var objVar;
...
objVar = tppcVar->get_typed_consumer();
```

`get_typed_consumer()` returns an object reference of type `CORBA::Object`. Therefore, the supplier must narrow this object reference to a reference of type `StockPrice`.

```
// C++
if (stockVar = StockPrice::_narrow (objVar)) {
   ...
else // call to _narrow() failed.
```

The supplier will use this object reference to push events to the event channel.

Pushing Events to an Event Channel

The following code extract from the example supplier program shows how the supplier initiates the transfer of events.

```
// C++
while (!tpsImpl.m_disconnected) {
   stockVar->quote (24.60);
}
```

In this example, the supplier repeatedly pushes an event to the event channel by calling the operation `quote()` on a `TypedProxyPushConsumer` object. The `TypedProxyPushConsumer` object implements the `StockPrice` interface that the supplier and consumer have agreed will be used to communicate typed events between them. The `quote()` operation takes one parameter of type `float` which contains the price of the stock item.

Note that our supplier stops sending events only when it receives an incoming `disconnect_push_supplier()` operation call from the event channel.

As an alternative, the supplier could explicitly disconnect from the event channel by invoking the operation `disconnect_push_consumer()` on the event channel `TypedProxyPushConsumer` object.

A Typed Push Supplier Application

The following source code implements a typed push supplier which supplies `StockPrice::quote()` events. It illustrates how the four programming steps described in detail in the preceding subsections fit in to a typed push supplier application.

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include "StockPrice_i.h"
...

int main(int argc, char** argv) {
   CosTypedEventChannelAdmin::TypedEventChannel_var tecVar;
   CosTypedEventChannelAdmin::TypedSupplierAdmin_var tsaVar;
   CosTypedEventChannelAdmin::TypedProxyPushConsumer_var tppcVar;
   CORBA::Object_var objVar;
   TypedPushSupplier_i tpsImpl;
   StockPrice_var stockVar("IONAY", 24);
   char *serverHost;

   try {
      //
      // Step 1. Get a TypedProxyPushConsumer object reference.
      //

      // Obtain an typed event channel reference.
      try {
         tecVar = TypedEventChannel::
            _bind ("Typed_Channel_1:ES", serverHost);
      }
      catch (...) {
         // Handle exception.
   ...
      }
      if (CORBA::is_nil (tecVar))
         return 1;

      // Obtain a supplier administration object.
      saVar = tecVar->for_suppliers ();
      // Obtain a typed proxy push consumer.
      tppcVar = saVar->obtain_typed_push_consumer ("StockPrice");

      //
      // Step 2. Connect a push supplier implementation object.
      //
      tppcVar->connect_push_supplier (&tpsImpl);

      //
      // Step 3. Obtain a typed push consumer object reference.
      //
      objVar = tppcVar->get_typed_consumer();

      if (stockVar = StockPrice::_narrow (objVar)) {
         //
         // Step 4. Push events to the event channel.
         //
         while (!tpsImpl.m_disconnected) {
            stockVar->quote (24.60);
            CORBA::Orbix.processNextEvent (1000);
         }
      } else cout << "Attempt to _narrow() failed." << endl;

      // When finished, disconnect the consumer.
      tppcVar->disconnect_push_consumer();
   }
   catch (...) {
      // Handle exception
      ...
      return 1;
   }
   return 0;
}
```

Developing a Typed Push Consumer

A typed push consumer receives events from an event channel, with no knowledge of the suppliers from which those events originated. The event channel, in turn, receives events from push suppliers in the form of operation invocations on the interface agreed between the suppliers and the event channel. An event channel propagates an event to a typed push consumer by invoking the operation it has received on a `TypedPushConsumer` implementation object in the consumer application. As such, the main functionality of a typed push consumer is associated with registering a `TypedPushConsumer` object with an event channel and receiving incoming operation calls on that object.

To develop a typed push consumer application, you must implement the following steps:

1. Obtain a reference for a `ProxyPushSupplier` object from the event channel.

   "Obtaining a ProxyPushSupplier from an Event Channel" on page 57 explains this step.
2. Connect a `TypedPushConsumer` implementation object to the event channel, by invoking the operation `connect_push_consumer()` on the `ProxyPushSupplier` object, passing an object of type `TypedPushConsumer` as an operation parameter.

   "Connecting a TypedPushConsumer Object to an Event Channel" on page 57 explains this step.
3. Monitor incoming operation calls.

   "Monitoring Incoming Operation Calls" on page 60 explains this step.

"A Typed Push Consumer Application" on page 61 show how these steps fit in to a full typed push consumer application.

Obtaining a ProxyPushSupplier from an Event Channel

Each typed push consumer connected to an event channel receives every event raised by every supplier connected to the channel. However, consumers have no knowledge of the suppliers. Consumers simply connect to an object in the event channel which acts as a single source of events.

This object is responsible for storing a `TypedPushConsumer` object reference for each connected consumer and propagating the operation invocation it receives on each of these references when a supplier transmits an event. The event channel object which stores consumer references is of type `ProxyPushSupplier`. The first task in developing a push consumer application is to obtain a reference to this object.

There are three stages in obtaining a `ProxyPushSupplier` object reference:

1. Obtain a reference to a `TypedEventChannel` object in the event channel.
2. Invoke the operation `for_consumers()` on the `TypedEventChannel` object, in order to obtain a `TypedConsumerAdmin` object reference.
3. Invoke the operation `obtain_typed_push_supplier()` on the `TypedConsumerAdmin` object and pass the name of the interface agreed between the event channel and the typed consumer as a parameter. This operation returns a `ProxyPushSupplier` object reference.

You may implement the first of these steps in exactly the manner described for typed push supplier applications in "Obtaining a TypedProxyPushConsumer from an Event Channel" on page 50. The remaining steps involve normal operation invocations.

Connecting a TypedPushConsumer Object to an Event Channel

When a typed consumer has obtained a reference to the `ProxyPushSupplier` object in an event channel, the next step is to register a `TypedPushConsumer` implementation object with the `ProxyPushSupplier`. The event channel uses the `TypedPushConsumer` object to propagate events to the consumer.

As described in Chapter 2, "The Programming Interface to the Event Service", the CORBA Event Service specification defines the interface TypedPushConsumer as follows:

```
// IDL
module CosTypedEventComm {
   interface TypedPushConsumer : CosEventComm::PushConsumer {
      Object get_typed_consumer ();
   };
   ...
};
```

When an event arrives at an event channel in the form of an invocation on any of the operations defined on the interface agreed between the supplier and consumer, the channel ProxyPushSupplier object propagates the operation call by invoking the same operation on each connected consumer. The disconnect_push_consumer() operation allows an event channel to disconnect a consumer, for example, if the channel closes down.

Our typed consumer uses the following example implementation of this interface:

```
// C++
class StockPrice_i : public virtual StockPriceBOAImpl {
   protected:
   CORBA::Float m_price;
   char* m_company;

   public:
   CORBA::Boolean m_disconnected;

   StockPrice_i (char* company,
      CORBA::Float price,
      CORBA::Environment& env = CORBA::default_environment){
      m_disconnected = 0;
      m_price = price;
      m_company = new char[strlen(company)+1];
      strcpy(m_name, company);
   }

   virtual void disconnect_push_consumer (
      CORBA::Environment& env = CORBA::default_environment){
      m_disconnected = 1;
   }
   virtual void push (CORBA::Any& any,
      CORBA::Environment& env = CORBA::default_environment){
      throw CORBA::NO_IMPLEMENT;
   }

   virtual CORBA::Object get_typed_consumer (
      CORBA::Environment& env = CORBA::default_environment){
      return this;
   }

   virtual void quote (CORBA::Float new_price,
      CORBA::Environment& env = CORBA::default_environment) {
      m_price = new_price;
      cout << endl
           << "Stock: " << m_company << "now at "
           << new_price << endl << endl;
   }
};
```

The `StockPrice` interface defined the operation, `quote()`. Class `StockPrice_i` implements this operation through which the consumer receives events.

Because `TypedPushConsumer` inherits from `PushConsumer`, it must provide an implementation of the `push()` operation defined on interface `PushConsumer`. In this example, class `StockPrice_i` provides a null implementation of `push()`, which simply raises the standard CORBA exception `CORBA::NO_IMPLEMENT`. This restricts suppliers to using typed communication with this consumer. Alternatively, class `StockPrice_i` could implement `push()` so that a supplier could use untyped as well as typed event communication with the consumer, passing a changed stock price in the `push()` operation's any parameter.

Monitoring Incoming Operation Calls

The main role of the typed consumer is to receive events from the event channel in the form of IDL operation calls. Consequently, the consumer must monitor and process any incoming calls. The example Orbix consumer application does this by repeatedly calling `processNextEvent()` on the `CORBA::Orbix` object, as follows:

```
// C++
while (!stockPriceImpl.m_disconnected) {
   CORBA::Orbix.processNextEvent ();
}
```

The function `processNextEvent()` handles a single incoming operation call and then returns.

If the consumer receives an invocation on the operation `disconnect_push_consumer()`, then the implementation of this operation sets the value `stockPriceImpl.m_disconnected` to one and breaks the consumer's event processing loop. Consequently, our consumer receives all events until the event channel explicitly forces it to disconnect.

As an alternative, the consumer could explicitly disconnect itself from the event channel when it no longer wishes to receive events. The consumer does this by invoking `disconnect_push_supplier()` on the event channel `ProxyPushSupplier` object.

A Typed Push Consumer Application

The three main programming steps in the development of a typed push consumer applications have been described in detail.

The following source code illustrates how each of these steps fits in to the full typed push supplier application. The application obtains a typed proxy push supplier for the interface `StockPrice` and then waits for events.

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include "TypedPushConsumer_i.h"
#include "StockPrice_i.h"

int main(int argc, char** argv) {
   CosTypedEventChannelAdmin::TypedEventChannel_var tecVar;
   CosTypedEventChannelAdmin::TypedConsumerAdmin_var tcaVar;
   CosTypedEventChannelAdmin::ProxyPushSupplier_var ppsVar;
   StockPrice_i stockPriceImpl;
   char *serverHost

   try {
      //
      // Step 1. Get a ProxyPushSupplier object reference.
      //

      // Obtain a typed event channel reference.
      try {
         tecVar = TypedEventChannel::_bind (
            "Typed_Channel_1:ES", serverHost);
      }
      catch (...) {
         // Handle exception.
   ...
      }
      if (CORBA::is_nil (tecVar))
         return 1;

      // Obtain a typed consumer administration object.
      tcaVar = tecVar->for_consumers ();
      // Obtain a typed proxy push supplier for
      // the interface StockPrice.
      ppsVar = tcaVar->obtain_typed_push_supplier ("StockPrice");

      //
      // Step 2. Connect a typed push consumer
      // implementation object.
      //
      ppsVar->connect_push_consumer (&stockPriceImpl);

      //
      // Step 3. Monitor incoming operation calls.
      //
      while (!stockPriceImpl.m_disconnected) {
         CORBA::Orbix.processNextEvent ();
      }

      // When finished, disconnect the supplier.
      tppsVar->disconnect_push_supplier();
   }
   catch (...) {
      // Handle exception.
      ...
      return 1;
   }
   return 0;
}
```

6

Programming with the Untyped Pull Model

To illustrate the Pull model to transfer untyped events, this chapter develops a simple application.

As described in Chapter 1, "Introduction to the CORBA Event Service", OrbixEvents allows you to develop Object Request Broker (ORB) applications that communicate using the CORBA Event Service communications model. From a programmer's perspective, the event channel is the key element of a CORBA Event Service application.

This chapter describes an example ORB application that illustrates how you can use OrbixEvents to develop pull model suppliers and consumers that communicate untyped events through event channels.

Overview of an Example Application

The example described in this chapter consists of a pull supplier and a pull consumer, each of which connects to a single event channel. The consumer repeatedly pulls an event from the event channel. The data associated with each event takes the form of a string, and the consumer simply displays the data as it receives it. The event channel, in turn, pulls the data from a pull supplier. This application is straightforward, but it illustrates a series of development tasks that apply to all OrbixEvents applications.

When developing an OrbixEvents application, you must implement the suppliers and consumers as normal ORB applications that communicate with the event channel through IDL interfaces. OrbixEvents fully implements the event channel, which is created in the OrbixEvents server application. The IDL definitions for the CORBA Event Service are supplied with OrbixEvents.

This chapter examines the implementation of a supplier and consumer using Orbix for C++ as the development ORB. However, the OrbixEvents server fully supports the CORBA Internet Inter-ORB Protocol (IIOP), so you may develop OrbixEvents applications using any IIOP-compatible ORB.

Developing an Untyped Pull Consumer

As described in "Transfer of Typed Events Through an Event Channel" on page 24, a pull consumer initiates the transfer of an event by requesting the event from the event channel. The event channel, if it does not already have an event to meet the request, requests an event from each registered supplier and then passes an event to the pull consumer. A pull consumer may poll for an event if it does not want to block while waiting for an event to become available.

To develop a pull consumer application, you must implement the following steps:

1. Obtain a `ProxyPullSupplier` object from the event channel.
2. Invoke the operation `connect_pull_consumer()` on the `ProxyPullSupplier` object, to connect a `PullConsumer` implementation object to the event channel.
3. Invoke `try_pull()` operations on the `ProxyPullSupplier` object to initiate the transfer of each event. (As an alternative you can also use the `pull()` operation. `try_pull()` is preferred however.)

Obtaining a ProxyPullSupplier from an Event Channel

A pull consumer connected to an event channel receives an event only when it explicitly requests one. The consumer has no knowledge of the suppliers; it simply connects to an object in the event channel that acts as a single source of events.

This object is responsible for storing a `PullSupplier` object reference for each connected supplier and invoking a `try_pull()` or `pull()` operation on each of these object references when a consumer requests an event using `try_pull()` or `pull()` respectively. The event channel object which stores supplier references is of type `ProxyPullSupplier`. The first task in developing a push consumer application is to obtain a reference to this object.

As illustrated in our example pull consumer application, a pull consumer obtains a reference to a `ProxyPullSupplier` by implementing the following steps:

1. Obtain a reference to an `EventChannel` object in the event channel.
2. Invoke the operation `for_consumers()` on the `EventChannel` object in order to obtain a `ConsumerAdmin` object.
3. Invoke the `operation obtain_pull_supplier()` on the `ConsumerAdmin` object. This operation returns a `ProxyPullSupplier` object reference.

You may implement the first of these steps exactly as described for push supplier applications in "Obtaining a ProxyPushConsumer from an Event Channel" on page 35. The remaining steps involve normal operation invocations.

Connecting a PullConsumer Object to an Event Channel

When the consumer has obtained a reference to a `ProxyPullSupplier` object from the event channel, it needs to connect an implementation of the `PullConsumer` interface to the event channel. As described in "The Pull Model for Untyped Events" on page 15, this interface is defined as follows:

```
// IDL
module CosEventComm {
   ...

   interface PullConsumer {
      void disconnect_pull_consumer ();
   };
};
```

The purpose of this interface is to allow the event channel to disconnect the `PullConsumer` by invoking the operation `disconnect_pull_consumer()`. This may be necessary if the event channel closes down.

In our example, the consumer application implements the `PullConsumer` interface by defining the class `PullConsumer_i` and implementing it as follows:

```
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
...

class PullConsumer_i
   : public virtual CosEventComm::PullConsumerBOAImpl {

   public:
   CORBA::Boolean m_disconnected;

   PullConsumer_i () {
      m_disconnected = 0;
   }

   void disconnect_pull_consumer (
      CORBA::Environment& env = CORBA::default_environment) {
      m_disconnected = 1;
   }
};
```

Class `PullConsumer_i` uses a simple flag mechanism to indicate the connection state of the consumer. The consumer connects an object of this type to an event channel by calling the operation `connect_pull_consumer()` on the `ProxyPullSupplier` object.

Pulling Events from an Event Channel

The following code extract from the example consumer program shows a simple way of initiating the transfer of events using the `try_pull()` operation:

```
CORBA::Boolean got_event;
...
while (!pcImpl.m_disconnected) {
   event = ppsVar->try_pull(got_event);
   if (got_event) {
      if (*event >>= eventDataString) {
         cout << eventDataString << endl;
         delete [] eventDataString;
         delete event;
      } else {
         cout << "Error: Pulled bad data" << endl;
      }
   } else {
      cout << "Event channel did not supply event" << endl;
   }
   CORBA::Orbix.processNextEvent (1000);
}
```

In this example, the consumer repeatedly pulls an event from the event channel using the `try_pull()` operation on a `ProxyPullSupplier` object in the channel. In this example, the event supplied in the any return value of the `try_pull()` operation is a string; in general, the type contained in this any is application dependent.

The `try_pull()` operation pulls events without blocking. The `pull()` operation causes the consumer application to block until a event is supplied by the channel. If you are using a multi-thread safe ORB such as Orbix-MT or OrbixWeb, you could also create an application thread dedicated to pulling events from the channel without blocking the consumer application. The following code extract illustrates the use of `pull()`:

```
// C++
while (!pcImpl.m_disconnected) {
   event = ppsVar->pull();
   if (*event >>= eventDataString) {
      cout << eventDataString << endl;
      delete [] eventDataString;
      delete event;
   } else {
      cout << "Error: Pulled bad data" << endl;
   }
   CORBA::Orbix.processNextEvent (1000);
}
```

The consumer stops pulling events only when it receives an incoming `disconnect_pull_consumer()` operation call from the event channel. Alternatively, the consumer could explicitly disconnect from the event channel by invoking the operation `disconnect_pull_supplier()` on the `ProxyPullSupplier` object in the event channel.

An Untyped Pull Consumer Application

The following source code illustrates the implementation of a simple pull consumer that pulls events using the `try_pull()` operation:

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include <PullConsumer_i.h>
...
int main(int argc, char** argv) {
   CosEventChannelAdmin::EventChannel_var ecVar;
   CosEventChannelAdmin::ConsumerAdmin_var caVar;
   CosEventChannelAdmin::ProxyPullSupplier_var ppsVar;
   CORBA::Any* event;
   char* eventDataString;
   PullConsumer_i pcImpl;
   CORBA::Boolean got_event;
   char *serverHost;


   try
      //
      // Step 1. Get a ProxyPullSupplier object reference.
      //

      // Obtain an event channel reference.
   try {
      ecVar = EventChannel::_bind ("Channel_1:ES",
         serverHost);
   }
   catch (...) {
      // Handle exception.
   ...
   }
      if (CORBA::is_nil (ecVar))
         return 1;

      // Obtain a consumer administration object.
      caVar = ecVar->for_consumers ();

      // Obtain a proxy pull supplier.
      ppsVar = saVar->obtain_pull_supplier ();

      //
      // Step 2. Connect a pull consumer implementation object.
      //
      ppsVar->connect_pull_consumer (&pcImpl);

      //
      // Step 3. Pull events from the event channel.
      //
      while (!pcImpl.m_disconnected) {
         event = ppsVar->try_pull(got_event);
         if (got_event){
            if(*event >>= eventDataString){
               cout << eventDataString << endl;
               delete[] eventDataString;
               delete event;
            } else {
               cout << "Error: Pulled bad data" << endl;
         else {
            cout << "Event channel did not supply event" << endl;

         }
         CORBA::Orbix.processNextEvent (1000);
      }
   }
   catch (...) {
      // Handle exception.
      ...
      return 1;
   }
   return 0;
}
```

"Obtaining a ProxyPullSupplier from an Event Channel" on page 65, "Connecting a PullConsumer Object to an Event Channel" on page 65, and "Pulling Events from an Event Channel" on page 67 explain the details of each step in the implementation of a `PullConsumer` with reference to this source code.

Developing an Untyped Pull Supplier

A pull supplier supplies events on request to an event channel and has no knowledge of the consumers to which these events will be propagated. The event channel requests an event from a pull supplier in order to fulfil a request for an event by a pull consumer. An event channel requests an event by invoking the `pull()` or `try_pull()` operations on a `PullSupplier` object in the supplier application. A supplier application, therefore, must register a `PullSupplier` object with an event channel and receive incoming operation calls on that object.

To develop a pull supplier application, you must implement the following steps:

1. Obtain a reference for a `ProxyPullConsumer` in the event channel.
2. Connect a `PullSupplier` implementation object to the event channel by invoking the operation `connect_pull_supplier()` on the `ProxyPullConsumer` object.
3. Monitor incoming operation calls.

Obtaining a ProxyPullConsumer from an Event Channel

When a pull consumer requests an event, the event channel to which it is connected in turn requests an event from each connected pull supplier if it does not already have an event stored in the channel. The suppliers have no knowledge of the consumers requesting events; they simply connect to an object in the event channel.

This object is responsible for storing a `PullSupplier` object reference for each connected supplier, and invoking the `pull()` or `try_pull()` operation on each of these references when a consumer requests an event. The event channel object which stores supplier references is of type `ProxyPullConsumer`. The first task in developing a pull supplier application is to obtain a reference to this object.

As illustrated in our example supplier source code, this reference is obtained by implementing the following steps:

1. Obtain a reference to an `EventChannel` object in the event channel.
2. Invoke the operation `for_suppliers()` on the `EventChannel` object in order to obtain a `SupplierAdmin` object.
3. Invoke the `operation obtain_pull_consumer()` on the `SupplierAdmin` object. This operation returns a `ProxyPullConsumer` object reference.

You may implement the first of these steps exactly as described for push supplier applications in "Obtaining a ProxyPushConsumer from an Event Channel" on page 35. The remaining steps involve normal operation invocations.

Connecting a PullSupplier Object to an Event Channel

When a supplier has obtained a reference to a `ProxyPullConsumer` object in an event channel, the next step is to register a `PullSupplier` implementation object with the `ProxyPullConsumer`.

As described in "The Pull Model for Untyped Events" on page 15, the CORBA Event Service specification defines the interface `PullSupplier` as follows:

```
// IDL
module CosEventComm {
   interface PullSupplier {
      any pull () raises (Disconnected);
      any try_pull (out boolean has_event) raises (Disconnected);
      void disconnect_pull_supplier();
   };
   ...
};
```

When a request for an event arrives at an event channel in the form of a `pull()` or `try_pull()` operation from a pull consumer, the channel `ProxyPullConsumer` object invokes a corresponding `pull()` or `try_pull()` operation on each connected supplier.

The disconnect_pull_supplier() operation allows the event channel to disconnect a supplier, for example, if the event channel closes down.

Our example supplier implements this interface as follows:

```
// C++
class PullSupplier_i :
   public virtual CosEventComm::PullSupplierBOAImpl {

   protected:
   unsigned char m_generate_event;

   public:
   CORBA::Boolean m_disconnected;

   PullSupplier_i () {
      m_disconnected = 0;
      m_have_event = 0;
   }

   virtual void disconnect_pull_supplier (
      CORBA::Environment& env = CORBA::default_environment){
      m_disconnected = 1;
   }

   virtual CORBA::Any* pull (
      CORBA::Environment& env = CORBA::default_environment){
      CORBA::Any a;
      char* eventDataString = "Hello World!";
      if (!m_disconnected) {
         a <<= eventDataString;
         return a;
      } else {
         throw CosEventComm::Disconnected;
         return 0;
      }
   }

   virtual CORBA::Any* try_pull (
      CORBA::Boolean& has_event,
      CORBA::Environment& env = CORBA::default_environment){
         // This trivial implementation of try_pull()
         // supplies an event on every alternate call.
         CORBA::Any a;
         char* eventDataString = "Hello World!";
         if (!m_disconnected) {
            if (m_generate_event) {
               a <<= eventDataString;
               m_has_event = 1;
               m_generate_event = 0;
               return a;
            }
            else {
               m_has_event = 0;
               m_generate_event = 1;
            }
         }
      } else {
         throw CosEventComm::Disconnected;
         return 0;
      }
   }
};
```

This class includes trivial implementations of the `pull()` and `try_pull()` operations which deal with requests for events. The exact requirements for implementing these operations are application specific; a real OrbixEvents application would probably require more complex implementations.

Monitoring Incoming Operation Calls

A pull supplier application receives requests for events from an event channel in the form of `pull()` and `try_pull()` operation calls; the event channel may also disconnect the supplier by invoking the operation `disconnect_pull_supplier()`. The supplier must, therefore, monitor and process incoming operation calls. The example pull supplier application does this by repeatedly calling `processNextEvent()` on the `CORBA::Orbix` object, as follows:

```
while (!psImpl.m_disconnected) {
   CORBA::Orbix.processNextEvent (1000);
}
```

The function processNextEvent() handles a single incoming operation call and then returns. This example uses a timeout value of 1000 milliseconds, but any finite value would be appropriate.

If the supplier receives a disconnect_pull_supplier() operation invocation, then the implementation of this operation sets the value psImpl.m_disconnected to one and breaks the supplier's event processing loop. In this way, our supplier receives operation invocations until the event channel explicitly asks it to disconnect. A supplier could explicitly disconnect itself from the event channel when it no longer wants to supply events, by invoking the operation disconnect_pull_consumer() on the event channel ProxyPullConsumer object.

An Untyped Pull Supplier Application

The following code implements an example pull supplier:

```
// C++
#include <CORBA.h>
#include <cosevents.hh>
#include <coseventsadmin.hh>
#include <PullSupplier_i.h>

int main(int argc, char** argv) {
   char *eventChannelName = "Channel_1"
   CosEventChannelAdmin::EventChannel_var ecVar;
   CosEventChannelAdmin::SupplierAdmin_var saVar;
   CosEventChannelAdmin::ProxyPullConsumer_var ppcVar;
   PullSupplier_i psImpl;

   try {
      //
      // Step 1. Get a ProxyPullConsumer object reference.
      //

      // Obtain an event channel reference.
   try {
      ecVar = EventChannel::_bind ("Channel_1:ES",
         serverHost);
   }
   catch (...) {
      // Handle exception.
   ...
   }
      if (CORBA::is_nil (ecVar))
         return 1;

      // Obtain a supplier administration object.
      saVar = ecVar->for_suppliers ();

      // Obtain a proxy pull consumer.
      ppcVar = saVar->obtain_pull_consumer ();

      //
      // Step 2. Connect a pull supplier implementation object.
      //
      ppcVar->connect_pull_supplier (&psImpl);

      //
      // Step 3. Monitor incoming operation calls.
      //
      while (!psImpl.m_disconnected) {
         CORBA::Orbix.processNextEvent (1000);
      }
   }
   catch (...) {
      // Handle exception.
      ...
      return 1;
   }
   return 0;
}
```

7

Compiling and Running an OrbixEvents Application

You will need to compile the IDL definitions for the Event Service as well as compile and build your application. This chapter describes how to do this.

Compiling the IDL Definitions for the Event Service

The CORBA standard IDL interfaces for CORBA Event Service suppliers, consumers and event channels are defined in the files `cosevents.idl`, `coseventsadmin.idl`, and `orbixevents.idl` in the `idl` directory of your OrbixEvents installation. These are the contents of the IDL files:

IDL File	Contents
`cosevents.idl`	This file contains the `CosEventComm` and `CosTypedEventComm` modules.
`coseventsadmin.idl`	This file contains the `CosEventChannelAdmin` and `CosTypedEventChannelAdmin` modules.
`orbixevents.idl`	This file contains the `OrbixEventsAdmin` module.

Table 7.1: *OrbixEvents IDL Files*

Although the programming steps required to develop a supplier or consumer depend on your development ORB, the first step generally involves using an IDL compiler to compile these definitions.

Using Orbix you can invoke the command-line IDL compiler as follows:

```
idl cosevents.idl,
idl coseventsadmin.idl
idl orbixevents.idl
```

This compilation command generates these C++ files:

`cosevents.hh`, `coseventsadmin.hh` `orbixevents.hh`	Header files that include the C++ view of the IDL definitions. These files must be included in all supplier and consumer applications.
`coseventsS.cpp`	A source file that includes both client stub code and server skeleton code for the IDL definitions. This file must be compiled and linked with each supplier and consumer application.
`coseventsadminC.cpp` `orbixeventsC.cpp`	Source files that include client stub code for the IDL definitions. These files must be compiled and linked with each supplier and consumer application.

The `.cpp` extension is only used in some environments; extensions such as `.cc` may also be produced.

Compiling an OrbixEvents Application

An OrbixEvents supplier or consumer application is simply a standard ORB application that communicates with an event channel server through a set of IDL interfaces. In addition, both suppliers and consumers implement IDL interfaces and therefore act as ORB servers.

To compile an OrbixEvents application, you should follow the server compilation steps associated with your development ORB. For example, the following steps are required to build an Orbix application that communicates with an OrbixEvents event channel:

1. Compile the IDL definitions accessed by your application, including those in the files `cosevents.idl`, `coseventsadmin.idl`, and `orbixevents.idl` as described in “Compiling the IDL Definitions for the Event Service” on page 77.
2. Compile any IDL generated C++ files required by your application, including the files `coseventsS.cpp`, `coseventsadminC.cpp`, and `orbixeventsC.cpp`.
3. Compile all other C++ source files associated with your application.
4. Link the object files from steps 2 and 3 with the appropriate Orbix libraries.

Running an OrbixEvents Application

OrbixEvents supplies a server that supports untyped events and typed events. These servers are implemented as Orbix server applications and consequently require an installation of Orbix or OrbixWeb. Before running an OrbixEvents application, you must first register the OrbixEvents server you want to use with the Orbix Implementation Repository.

The OrbixEvents Server

The OrbixEvents server is implemented by the application `es` in the `bin` directory of your OrbixEvents installation. In its most basic form, the command line for this application appears as follows:

```
es channel_id
```

By default, the channels created in this way are for untyped events. You can specify a list of typed event channels with the command-line argument `typed_events` (see “The OrbixEvents Server Command Lines” on page 84). The `channel_id` is an identifier for an event channel implemented in the server. An OrbixEvents server may implement several event channels. In this case, each event channel must have an associated channel identifier and the list of identifiers that the server implements immediately follows the `es` command. For example, you may specify multiple event channels as follows:

```
es channel_id_1 channel_id_2 ...
```

Event channels can have any name, but cannot be prefixed by `otrmp//`, `otsfp//` or `otmcp//` as these indicate multicast event channels that are supported by the OrbixTalk 3.0 Gateway.

By default, you must register the OrbixEvents untyped events server with the Implementation Repository using the server name `ES`. For example, to register a server that implements a single event channel named `Channel_1`, you can use the Orbix `putit` command as follows[1]:

```
putit ES "/home/events/bin/es Channel_1"
```

The exact registration command depends on your application requirements. However, you must register the server in shared activation mode and you must not specify the `-n`, `-per-client` or `-per-client-pid` switches to `putit` during the registration.

Running your Application

Once you have registered the OrbixEvents server, you can run your supplier and consumer applications. In the examples in Chapter 4, “Programming with the Untyped Push Model”, Chapter 5, “Programming with the Typed Push Model”, and Chapter 6, “Programming with the Untyped Pull Model” the order in which you run the consumer and supplier applications has no effect on the system functionality. You do not need to register the example suppliers or the example consumers in this guide in the Orbix Implementation Repository.

Lifetime of Proxy Objects

The event server creates a new proxy object when requested for one. This object persists until:

1. Disconnect is invoked upon it.
2. The event channel is destroyed.
3. The IIOP connection is closed.
4. An attempt by the `(Typed)ProxyPushSuppliers` to `get_typed_consumer()` fails.

1. You can, alternatively, use the Orbix Server Manager GUI tool to register an OrbixEvents server.

The proxy is destroyed in all these cases. It is not possible perform another invocation on the object after that, including `push()`, `pull()`, `try_pull()`, `connect()`, or `disconnect()`. If an attempt is made to perform an operation on the destroyed proxy, an `INVALID OBJECT REFERENCE` exception is thrown.

If a `PullConsumer` has invoked `pull()` upon a `ProxyPullSupplier`, and meanwhile `disconnect_pull_supplier()` is invoked upon the `ProxyPullSupplier`, the `pull()` throws a `Disconnected` exception some time after (depending on the `pull_prod_interval` configuration value).

If you attempt to connect an invalid object to a proxy object (where an exception other than `INVALID OBJECT REFERENCE` is thrown), the proxy is not destroyed.

8

OrbixEvents Configuration

An OrbixEvents server must be available at each host where event channels are required. This chapter describes how to register an OrbixEvents server and how to create event channels within the server.

OrbixEvents is implemented as a single Orbix server application supporting both untyped and typed events.

To create an application that communicates with OrbixEvents, you develop a normal ORB application which acts as a client to the appropriate OrbixEvents server. Consequently, OrbixEvents imposes minimal configuration requirements beyond those of the ORB used to develop your application.

As described in Chapter 7, "Compiling and Running an OrbixEvents Application", the OrbixEvents server is implemented by the application `es` in the `bin` directory of your Orbix installation.

To make the OrbixEvents server available, you need to register the server with the Orbix Implementation Repository. For example, using the Orbix `putit` utility you could register the server as follows:

```
putit ES server_launch_command
```

To register an OrbixEvents server, you need to specify a launch command. "The OrbixEvents Server Command Lines" on page 84 describes the server command line in detail.

You can create event channels within an OrbixEvents server by specifying the channel identifiers in the server's command line.

"Assigning Identifiers to Event Channels" on page 87 describes how to assign channel identifiers.

You can configure many basic OrbixEvents settings using the Orbix configuration tool. Appendix B, "Configuration File Settings", details these configuration file variables.

The OrbixEvents Server Command Lines

The server's command line allow you to configure how the server is launched. It takes the following form:

```
es [-parameter | channel name]
```

Where the `parameter` is one of the command-line parameters listed in Table 8.1.

Each event channel implemented by the server must have an associated channel identifier. The list of channel identifiers that the server implements immediately follows the `es` command. "Assigning Identifiers to Event Channels" on page 87 describes the role of event channel identifiers in the OrbixEvents server and how these identifiers allow you to establish initial contact with an event channel.

The `es` command takes the following optional command-line switches:

Command-line Parameter	Effect
`untyped_channels`	Subsequent channel names are treated as untyped. This is the default.
`typed_channels`	Subsequent channel names are treated as typed.
`not_orbix_server`	Do not call `CORBA::Orbix.setServerName()` or `CORBA::Orbix.impl_is_ready()`.
`server_name`	The name passed to `CORBA::Orbix.setServerName()` and `CORBA::Orbix.impl_is_ready()`. Default is "ES".

Table: 8.1: *Event Server Command-line Options*

Command-line Parameter	Effect
srv_timeout	Millisecond time passed to single call to processEvents(). Default is INFINITE.
default_tx_timeout	Millisecond time passed to CORBA::Orbix.defaultTxTimeout(). Default is INFINITE.
use_transient_port	CORBA::Orbix.useTransientPort(1) is called.
set_diagnostics	Value of 0, 1, 2 or 3 passed to CORBA::Orbix.setDiagnostics(). Default is 0.
robust	Invocations to destroy on typed or untyped events channel will fail if any Proxies exist.
always_try_pull_on_suppliers	If a PullConsumer calls pull() on a ProxyPullSupplier then, by default, all ProxyPullConsumers call pull() on their connected PullSuppliers. If this parameter is set then ProxyPullConsumers call try_pull() on their connected PullSuppliers. Note that if a PullConsumer calls try_pull() on a ProxyPullSupplier then all ProxyPullConsumers will always call try_pull() on their connected PullSuppliers.
pull_prod_interval	When a PullConsumer calls pull() on a ProxyPullSupplier, this parameter sets the interval between: New ProxyPullConsumers calling pull() on their connected PullSuppliers. All ProxyPullConsumers calling try_pull() on their connected PullSuppliers.

Table: 8.1: *Event Server Command-line Options*

Command-line Parameter	Effect
try_pull_duration	When a PullConsumer calls try_pull() on a ProxyPullSupplier then this is the duration that try_pull() blocks, awaiting an event, before returning with has_event set to FALSE.
-nonames	Do not place the name OrbixEventsAdminChannelManager in root context of Name Service. (Default places it in root context if Name Service is running.)
I or i	OrbixEventsAdmin IOR is written to a file OrbixEventsAdmin.ref.
v	Version information and opal configuration values.
?	Usage information.
D	Dump current configuration settings.

Table: 8.1: *Event Server Command-line Options*

As described in Chapter 3, "OrbixEvents", an OrbixEvents server maintains a channel manager object to manage the event channels within the OrbixEvents server. Your application can invoke operations on the channel manager object, for example, to create and destroy event channels within the OrbixEvents server.

The channel manager object supports the interface OrbixEventsAdmin::ChannelManager. This interface is described in Appendix C, "OrbixEventsAdmin::ChannelManager".

You can use _bind() to obtain the OrbixEventsAdmin::ChannelManager object reference as follows:

```
// C++
OrbixEventsAdmin::ChannelManager_var cm;
char* serverHost;
...
try {
   channelManagerVar =
      OrbixEventsAdmin::ChannelManager::_bind(
                              "ChannelManager:ES", serverHost);
}
catch (...) {
   // Handle exception.
   ...
}
```

Assigning Identifiers to Event Channels

Each event channel in an OrbixEvents server has an associated channel identifier, which is unique within that server. You can specify the identifiers of the event channels that an OrbixEvents server implements in one of two ways:

1. As arguments to the server command line (see "The OrbixEvents Server Command Lines" on page 84).
2. As a parameter to the `OrbixEventsAdmin::ChannelManager::create()` operation (see Appendix C, "OrbixEventsAdmin::ChannelManager").

An event channel identifier is a string value that takes the form:

`[context{.context}:]channel_name`

An event channel can have any name, but cannot be prefixed by `otrmp//`, `otsfp//`, or `otmcp//` as these indicate multicast channels that are supported by the OrbixTalk Gateway.

The server uses the channel identifier as the marker value for a (typed or untyped) `EventChannel` object associated with the channel. The name associated with an `EventChannel` object consists of a sequence of name components corresponding to each component of the channel identifier.

Appendix A
Event Service IDL Definitions

This appendix lists the IDL definitions in the CORBA Event Service modules as specified by the CORBA standard.

OrbixEvents implements the definitions in the modules listed in the files `cosevents.idl`, `coseventsadmin.idl`, and `orbixevents.idl` in the `idl` directory of your OrbixEvents installation.

CosEvents.idl File Contents

The CosEventComm Module

```
// IDL

module CosEventComm
{

   exception Disconnected { };

  interface PushConsumer
  {
    void push (in any data) raises (Disconnected);
    void disconnect_push_consumer ();
  };

  interface PushSupplier
  {
    void disconnect_push_supplier();
  };

  interface PullSupplier
  {
    any pull () raises (Disconnected);
    any try_pull (out boolean has_event) raises (Disconnected);
    void disconnect_pull_supplier();
  };

  interface PullConsumer
  {
    void disconnect_pull_consumer();
  };
};
```

The CosTypedEventComm Module

```
//IDL
module CosTypedEventComm
{

  interface TypedPushConsumer : CosEventComm::PushConsumer
  {
    Object get_typed_consumer();
  };

  interface TypedPullSupplier : CosEventComm::PullSupplier
  {
    Object get_typed_supplier();
  };
};
```

CosEventsAdmin.idl File Contents

The CosEventChannelAdmin Module

```
//IDL
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 ();
   };

};
```

The CosTypedEventChannelAdmin Module

```
module CosTypedEventChannelAdmin
{

  exception InterfaceNotSupported {};
  exception NoSuchImplementation {};
  typedef string Key;

  interface TypedProxyPushConsumer :
    CosEventChannelAdmin::ProxyPushConsumer,
    CosTypedEventComm::TypedPushConsumer {};

  interface TypedProxyPullSupplier :
    CosEventChannelAdmin::ProxyPullSupplier,
    CosTypedEventComm::TypedPullSupplier {};

  interface TypedSupplierAdmin :
    CosEventChannelAdmin::SupplierAdmin
  {
    TypedProxyPushConsumer obtain_typed_push_consumer
    (
      in Key supported_interface
    ) raises(InterfaceNotSupported);

    CosEventChannelAdmin::ProxyPullConsumer
obtain_typed_pull_consumer
    (
      in Key uses_interface
    ) raises(NoSuchImplementation);
  };

  interface TypedConsumerAdmin :
   CosEventChannelAdmin::ConsumerAdmin
{
   TypedProxyPullSupplier obtain_typed_pull_supplier
   (
   in Key supported_interface
   ) raises(InterfaceNotSupported);

   CosEventChannelAdmin::ProxyPushSupplier
      obtain_typed_push_supplier
   (
   in Key uses_interface
   ) raises(NoSuchImplementation);
};

interface TypedEventChannel
{
   TypedConsumerAdmin for_consumers();
   TypedSupplierAdmin for_suppliers();
   void destroy();
   };
};
```

OrbixEvents.idl File Contents

The OrbixEventsAdmin Module

```
//IDL
module OrbixEventsAdmin
{
  exception duplicateChannel{ };   // Channel already exists
  exception noSuchChannel{ };     // Invalid channel or channel
                              //doesn’t exist

  interface ChannelManager
  {
    typedef sequence<string> stringSeq;

    CosEventChannelAdmin::EventChannel create
    (
      in string  channel_id
    ) raises (duplicateChannel);

    CosEventChannelAdmin::EventChannel find
    (
      in string channel_id
    ) raises (noSuchChannel);

    string findByRef
    (
      in CosEventChannelAdmin::EventChannel channel_ref
    )
    raises (noSuchChannel);

    stringSeq list();

    CosTypedEventChannelAdmin::TypedEventChannel createTyped
    (
      in string  channel_id
    ) raises (duplicateChannel);

    CosTypedEventChannelAdmin::TypedEventChannel findTyped
    (
      in string channel_id
    ) raises (noSuchChannel);

    string findByTypedRef
    (
      in CosTypedEventChannelAdmin::TypedEventChannel channel_ref
    )
    raises (noSuchChannel);

    stringSeq listTyped();
  };
};
```

Appendix B
Configuration File Settings

The OrbixEvents configuration file variables are contained in the scope OrbixOTM.OrbixEvents. You can adjust these settings with the Orbix configuration tool.

Variable	**Effect**
`IT_BATCH_SIZE`	Transmission batch size. The number of fragments sent per batch. Increasing this may increase the throughput for large messages. Default is 10
`IT_EVENTS_NOT_ORBIX_SERVER`	When this variable is set to `YES`, then OrbixEvents does not call `impl_is_ready()`. Default is `NO`. For example: `IT_EVENTS_NOT_ORBIX_SERVER = "NO";`
`IT_EVENTS_PULL_PROD_TYPE`	When a `PullConsumer` executes a `pull()` on the `ProxyPullSupplier` provided by the event server, this variable determines whether to attempt `pull()` or `try_pull()` on any `PullSuppliers` connected. Default is `PULL`, and the alternative is `TRY_PULL`. For example: `IT_EVENTS_PULL_PROD_TYPE = "PULL";`

Table B.1: *OrbixEvents Configuration Variables*

Variable	Effect
IT_EVENTS_PULL_PROD_INTERVAL	This value sets the interval in milliseconds between each attempted `try-pull()` or `pull()` on connected `PullSuppliers`. Default is `1000`. For example: `IT_EVENTS_PULL_PROD_INTERVAL = "1000";`
IT_EVENTS_SERVER_NAME	Server name used in call to `impl_is_ready()`. Default is `ES`. For example: `IT_EVENTS_SERVER_NAME = "ES";`
IT_EVENTS_TRY_PULL_DURATION	Determines how long in milliseconds a `PullConsumer` waits for an event after executing a `try_pull()` on the `ProxyPullSupplier` provided by the event server. Default is `100`. For example: `IT_EVENTS_TRY_PULL_DURATION = "100";`
IT_DEFAULT_TX_TIMEOUT	Timeout value in milliseconds passed to `defaultTxTimeout()`. Default is infinite. For example: `IT_DEFAULT_TX_TIMEOUT = 60000;`
IT_INITIAL_TYPED_EVENT_CHANNELS	This variable causes OrbixEvents to create typed events channels created at start-up with the channel name provided. Default is "". For example: `IT_INITIAL_TYPED_EVENT_CHANNELS = test_channel;`

Table B.1: *OrbixEvents Configuration Variables*

Variable	Effect
`IT_INITIAL_UNTYPED_EVENT_CHANNELS`	This variable causes OrbixEvents to create untyped events channels created at start-up with the channel name provided. Default is "". For example: `IT_INITIAL_UNTYPED_EVENT_CHANNELS = test_channel;`
`IT_LOG_CONSOLE`	Console log output flag. Sets whether logging will be shown on the console. Set to 0 if you do not wish logging to be sent to the console. Default is 1
`IT_LOG_FLAGS`	logging level flags. This allows configurable logging for certain parts of the EventServer. Flags include: "`USR, EVT, ERR, WARN, INFO, EVD, ITF, DLV, SFP, EVD, RMP, DEP, DB, FRAG, CPT, IMA, FT, TIM`" Default is `USR,EVT,ERR`
`IT_LOG_LEVEL`	System logging output level. The level determines the amount System logging information displayed. For example: `IT_LOG_LEVEL = 9;` 0 is none. 16 is high. 32 is maximum. Default is 0
`IT_LOG_SYSLOG`	System log file output flag.When set to 1 logs will output to file of the form `<app name>.<YYMMDD>_<HH.MM.SS>.pid<NNN>.txt` Default is 0
`IT_LOG_TID`	System logging show thread id flag. Default is 0

Table B.1: *OrbixEvents Configuration Variables*

Variable	Effect
`IT_MAX_MSG_SIZE_KB`	Max kilobytes message size. Sets a limit to the size of messages that can be sent. If you attempt to send a message that exceeds this limit, an exception is raised, and the message is not sent. Default is 200
`IT_MAX_PEND_KB`	Maximum Kilobytes waiting to be sent. Sets the size limit of the pending message queue. Messages are added to the pending queue by a push or pull operation. Messages are removed from the queue by the flow control mechanism as it sends message fragments. Once the queue becomes full, remote method invocations block until space becomes available in the queue. Default is 1280
`IT_MAX_RECV_KB`	Maximum kilobytes queued at receiver. When a consumer is executing code other than the Orbix event loop, messages arrive and are queued at the receiver. They are not dispatched to the appropriate user function until Orbix processes events in its event loop. The limit to the amount of data queued at a consumer is set with this variable. Default is 1280.

Table B.1: *OrbixEvents Configuration Variables*

Variable	Effect
`IT_MAX_SENT_KB`	Maximum kilobytes retained for retries. Specifies the upper limit to the sent message queue size in Kilobytes. Once the limit is reached, the oldest fragments are removed to make space as required for the new message fragments to be added to the queue. Default is 1280
`IT_ROBUST_EVENT_CHANNELS`	When this variable is set to `YES`, event channels are not destroyed by calls to `destroy()` if Proxies exist. Default is `NO`. For example: `IT_ROBUST_EVENT_CHANNELS = "NO";`
`IT_SERVER_TIMEOUT`	Timeout value in milliseconds passed to `processEvents()`. Default is infinite. `IT_SERVER_TIMEOUT = 60000;`
`IT_SET_DIAGNOSTICS`	Value passed to `setDiagnostics()`. Default is `1`. Valid values are `0`, `1` and `2`. For example: `IT_SET_DIAGNOSTICS = "0";`
`IT_USE_TRANSIENT_PORT`	When this variable is set to `YES`, OrbixEvents calls `useTransientPort(1)`. Default is `NO`. For example: `IT_USE_TRANSIENT_PORT = "NO";`
`IT_WRITE_IOR`	When this variable is set to `YE"`, `OrbixEventsAdmin IOR` is written to the file `OrbixEventsAdmin.ref`. Default is `NO`. For example: `IT_WRITE_IOR = "NO";`

Table B.1: *OrbixEvents Configuration Variables*

Appendix C OrbixEventsAdmin::ChannelManager

OrbixEvents provides the event channel administration interface, ChannelManager, defined in the module OrbixEventsAdmin, to allow you to create and manipulate multiple event channels within an OrbixEvents server.

The ChannelManager object in an OrbixEvents server is named "ChannelManager". You can obtain a reference to the ChannelManager by using _bind().

Synopsis

```
// IDL
module OrbixEventsAdmin {

    exception duplicateChannel{ };
    exception noSuchChannel{ };

    interface ChannelManager
    {
      typedef sequence<string> stringSeq;

      CosEventChannelAdmin::EventChannel create (
        in string  channel_id) raises
        (duplicateChannel);

      CosEventChannelAdmin::EventChannel find(
        in string channel_id)
        raises (noSuchChannel);

      string findByRef(
      in CosEventChannelAdmin::EventChannel
        channel_ref)
        raises (noSuchChannel);

      stringSeq list();

      CosTypedEventChannelAdmin::TypedEventChannel
        createTyped(
        in string  channel_id)
```

```
          raises (duplicateChannel);

        CosTypedEventChannelAdmin::TypedEventChannel
          findTyped(
          in string channel_id)
          raises (noSuchChannel);

        string findByTypedRef(
      in CosTypedEventChannelAdmin::TypedEventChannel
        channel_ref)
        raises (noSuchChannel);

        stringSeq listTyped();
      };
      };
```

ChannelManager::create()

Synopsis

```
CosEventChannelAdmin::EventChannel create (
  in string channel_id,
    raises duplicateChannel);
```

Description Creates an event channel.

Parameters

channel_id	The channel identifier for the event channel. The exception duplicateChannel is raised if the channel identifier specified in channel_id names an existing channel. "Assigning Identifiers to Event Channels" on page 87 describes the format of channel identifiers.

ChannelManager::createTyped()

Synopsis

```
CosTypedEventChannelAdmin::TypedEventChannel createTyped(
  in string  channel_id)
    raises (duplicateChannel);
```

Description Creates a Typed Event Channel

Parameters

Channel_id	The channel identifier for the typed event channel. The exception `duplicatedChannel` is raised if the channel identifier specified in `channel_id` names an existing channel.

ChannelManager::find()

Synopsis

```
CosEventChannelAdmin::EventChannel find (
   in string channel_id)
      raises (noSuchChannel);
```

Description Finds the event channel associated with the channel identifier `channel_id`.

Parameters

`channel_id`	The channel identifier for the event channel. The exception `noSuchChannel` is raised if the channel identifier specified in `channel_id` does not exist. “Assigning Identifiers to Event Channels” on page 87 describes the format of channel identifiers.

ChannelManager::findByRef()

Synopsis

```
string findByRef (
   in CosEventChannelAdmin::EventChannel channel_ref)
      raises (noSuchChannel);
```

Description Finds the channel identifier of the event channel specified in `channel_ref`.

Parameters

`channel_ref`	The object reference for the event channel. If `channel_ref` does not exist within the event server, `findByRef()` raises the exception `noSuchChannel`.

ChannelManager::findByTypedRef()

Synopsis

```
string findByTypedRef(
   in CosTypedEventChannelAdmin::TypedEventChannel channel_ref )
      raises (noSuchChannel);
```

Description Finds the channel identifier of the typed event channel specified in `channel_ref`.

Parameters

channel_ref	The object reference of the event channel. If `channel_ref` does not exist within the event server, `findByTypedRef` raises the exception `noSuchChannel`.

ChannelManager::findTyped()

Synopsis

```
CosTypedEventChannelAdmin::TypedEventChannel findTyped(
   in string  channel_id)
      raises (noSuchChannel);
```

Description Finds the typed event channel associated with the channel identifier `channel_id`.

Parameters

channel_id	The channel identifier for the typed event channel. The exception `noSuchChannel` is raised if the channel identifier specified in `channel_id` does not exist.

ChannelManager::list()

Synopsis

```
stringSeq list ();
```

Description Lists the event channels contained within the channel manager's event server.

ChannelManager::listTyped()

Synopsis

```
stringSeq listTyped();
```

Description Lists the `Typed Event Channel` contained within the Channel Manager's event server.

Index