Event Service Specification 4

4.1 Service Description

4.1.1 Overview

A standard CORBA request results in the synchronous execution of an operation by an object. If the operation defines parameters or return values, data is communicated between the client and the server. A request is directed to a particular object. For the request to be successful, both the client and the server must be available. If a request fails because the server is unavailable, the client receives an exception and must take some appropriate action.

In some scenarios, a more decoupled communication model between objects is required. For example:

- A system administration tool is interested in knowing if a disk runs out of space. The software managing a disk is unaware of the existence of the system administration tool. The software simply reports that the disk is full. When a disk runs out of space, the system administration tool opens a window to inform the user which disk has run out of space.
- A property list object is associated with an application object. The property list object is physically separate from the application object. The application object is interested in the changes made to its properties by a user. The properties can be changed without involving the application object. That is, in order to have reasonable response time for the user, changing a property does not activate the application object. However, when the application object is activated, it needs to know about the changes to its properties.
- A CASE tool is interested in being notified when a source program has been modified. The source program simply reports when it is modified. It is unaware of the existence of the CASE tool. In response to the notification, the CASE tool invokes a compiler.

- Several documents are linked to a spreadsheet. The documents are interested in knowing when the value of certain cells have changed. When the cell value changes, the documents update their presentations based on the spreadsheet. Furthermore, if a document is unavailable because of a failure, it is still interested in any changes to the cells and wants to be notified of those changes when it recovers.

4.1.2 Event Communication

The Event Service decouples the communication between objects. The Event Service defines two roles for objects: the supplier role and the consumer role. *Suppliers* produce event data and *consumers* process event data. Event data are communicated between suppliers and consumers by issuing standard CORBA requests.

There are two approaches to initiating event communication between suppliers and consumers, and two orthogonal approaches to the form that the communication can take.

The two approaches to initiating event communication are called the *push model* and the *pull model*. The push model allows a supplier of events to initiate the transfer of the event data to consumers. The pull model allows a consumer of events to request the event data from a supplier. In the push model, the supplier is taking the initiative; in the pull model, the consumer is taking the initiative.

The communication itself can be either generic or typed. In the generic case, all communication is by means of generic `push` or `pull` operations that take a single parameter that packages all the event data. In the typed case, communication is via operations defined in OMG IDL. Event data is passed by means of the parameters, which can be defined in any manner desired. Section 4.2 through section 4.5 discuss generic event communication in detail; section 4.6 through section 4.9 discuss typed event communication in detail.

An *event channel* is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a supplier of events. Event channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests.

4.1.3 Example Scenario

This section provides a general scenario that illustrates how the Event Service can be used.

The Event Service can be used to provide “change notification”. When an object is changed (its state is modified), an event can be generated that is propagated to all interested parties. For example, when a spreadsheet cell object is modified, all compound documents which contain a reference (link) to that cell can be notified (so the document can redisplay the referenced cell, or recalculate values that depend on

the cell). Similarly, when an engineering specification object is modified, all engineers who have registered an interest in the specification can be notified that the specification has changed.

In this scenario, objects that can be "changed" act as suppliers, parties interested in receiving notifications of changes act as consumers, and one or more event channel objects are used as intermediaries between consumers and suppliers. *Either the push or the pull model can be used at either end.*

If the push model is used by suppliers, objects that can be changed support the *PushSupplier* interface so that event communication can be discontinued, use the *EventChannel*, the *SupplierAdmin* and the *ProxyPushConsumer* interfaces to register as suppliers of events, and use the *ProxyPushConsumer* interface to push events to event channels.

When a change occurs to an object, a changeable object invokes a `push` operation on the channel. It provides as an argument to the `push` operation information that describes the event. This information is of data type `any` - it can be as simple or as complex as is necessary. For example, the event information might identify the object reference of the object that has been changed, it might identify the kind of change that has occurred, it might provide a new displayable image of the changed object or it might identify one or more additional objects that describe the change that has been made.

If the pull model is used by consumers, all client objects that want to be notified of changes support the *PullConsumer* interface so communication can be discontinued, using the *EventChannel*, *ConsumerAdmin* and *ProxyPullSupplier* interfaces to register as consumers of events, and using the *ProxyPullSupplier* interface to pull events from event channels.

The consumer may use either a blocking or non-blocking mechanism for receiving notification of changes. Using the `try_pull` operation, the consumer can periodically poll the channel for events. Alternatively, the consumer can use the `pull` operation which will block the consumer's execution thread until an event is generated by some supplier.

Event channels act as the intermediaries between the objects being changed and objects interested in knowing about changes. The channels that provide change notification can be general purpose, well-known objects (e.g., "persistent server-based objects" that are run as part of a workgroup-wide framework of objects that provide "desktop services") or specific-to-task objects (e.g., temporary objects that are created when needed). Objects that use event channels may locate the channels by looking for them in a persistently available server (e.g., by looking for them in a naming service) or they may be given references to these objects as part of a specific-to-task object protocol (e.g., when an "open" operation is invoked on an object, the object may return the reference to an event channel which the caller should use until the object is closed).

Event channels determine how changes are propagated between suppliers and consumers, i.e., the qualities of service (Section 4.1.6). For example, an event channel determines the persistence of an event. The channel may keep an event for a specified period of time, passing it along to any consumer who registers with the channel during

that period of time (e.g., it may keep event notifications about changes to engineering specifications for a week). Alternatively, the channel may only pass on events to consumers who are currently waiting for notification of changes (e.g., notifications of changes to a spreadsheet cell may only be sent to consumers who are currently displaying that cell).

This scenario exemplifies one way the event service described here forms a basic building block used in providing higher-level services specific to an application or common facilities framework of objects.

Instead of using the generic event channel, a typed event channel could also have been used.

4.1.4 Design Principles

The Event Service design satisfies the following principles:

- Events work in a distributed environment. The design does not depend on any global, critical, or centralized service.
- Event services allow multiple consumers of an event and multiple event suppliers.
- Consumers can either request events or be notified of events, whichever is more appropriate for application design and performance.
- Consumers and suppliers of events support standard OMG IDL interfaces; no extensions to CORBA are necessary to define these interfaces.
- A supplier can issue a single standard request to communicate event data to all consumers at once.
- Suppliers can generate events without knowing the identities of the consumers. Conversely, consumers can receive events without knowing the identities of the suppliers.
- The Event Service interfaces allow multiple qualities of service, for example, for different levels of reliability. It also allows for future interface extensions, such as for additional functionality.
- The Event Service interfaces are capable of being implemented and used in different operating environments, for example, in environments that support threading and those that do not.

4.1.5 Resolution of Technical Issues

This specification addresses the issues identified for event services in the OMG *Object Services Architecture*[1] document as follows:

1. *Object Services Architecture*, Document Number 92-8-4, Object Managment Group, Framingham, MA, 1992.

- **Distributed environment:** The interfaces are designed to allow consumers and suppliers of events to be disconnected from time to time, and do not require centralized event identification, processing, routing, or other services that might be a bottleneck or a single point of failure.

  Events themselves are *not* objects because the CORBA distributed object model does not support passing objects by value.

Event generation: The specification describes how events are generated and delivered in a very general fashion, with event channels as intermediate routing points. It does not require (or preclude) polling, nor does it require that an event supplier directly notify every interested party.

Events involving multiple objects: Complex events may be handled by constructing a notification tree of event consumer/suppliers checking for successively more specific event predicates. The specification does not require a general or global event predicate evaluation service as this may not be sufficiently reliable, efficient, or secure in a distributed, heterogeneous (potentially decoupled) environment.

Scoping, grouping, and filtering events: The specification takes advantage of CORBA's distributed scoping and grouping mechanisms for the identifier and type of events. Event filtering is easily achieved through event channels that selectively deliver events from suppliers to consumers. Event channels can be composed; that is, one event channel can consumer events supplied by another.

Typed event channels can provide filtering based on event type.

Registration and generation of events: Consumers and suppliers register with event channels themselves. Event channels are objects and they are found by any fashion that objects can be found. A global registration service is not required; any object that conforms to the IDL interface may consume an event.

Event parameters: The specification supports a parameter of type `any` that can be delivered with an event, used for application-specific data.

Forgery and secure events: Because event suppliers are objects, the specification leverages any ORB work on security for object references and communication.

Performance: The design is a minimalist one, and requires only one ORB call per event received. It supports both push-style and pull-style notification to avoid inefficient event polling. Since event suppliers, consumers, and channels are all ORB objects, the service directly benefits from a Library Object Adapter or any other ORB optimizations.

Formalized Event Information: For specific application environments and frameworks it may be beneficial to formalize the data associated with an event (defined in this specification as type any). This can be accomplished by defining a typed structure for this information. Depending on the needs of the environment, the kinds of information included might be a priority, timestamp, origin string, and confirmation indicator. This information might be solely for the benefit of the event consumer or might also be interpreted by particular event channel implementations.

Confirmation of Reception: Some applications may require that consumers of an event provide an explicit confirmation of reception back to the supplier. This can be supported effectively using a “reverse” event channel through which consumers send back confirmations as normal events. This obviates the need for any special confirmation mechanism. However, strict atomic delivery between all suppliers and all consumers requires additional interfaces.

4.1.6 Quality of Service

Application domains requiring event-style communication have diverse reliability requirements, from “at-most-once” semantics (best effort) to guaranteed “exactly-once” semantics, availability requirements, throughput requirements, performance requirements (i.e., how fast events are disseminated), and scalability requirements.

Clearly no single implementation of the Event Service can optimize such a diverse range of technical requirements. Hence, multiple implementations of event services are to be expected, with different services targeted toward different environments. As such, the event interfaces do not dictate *qualities of service*. Different implementations of the Event Service interfaces can support different qualities of service to meet different application needs.

For example, an implementation that trades at most once delivery to a single consumer in favor of performance is useful for some applications; an implementation that favors performance but cannot preclude duplicate delivery is useful for other applications. Both are acceptable implementations of the interfaces described in this chapter.

Clearly, an implementation of an event channel that discards all events is *not a useful* implementation. Useful implementations will at least support “best-effort” delivery of events.

Note that the interfaces defined in this chapter are incomplete for implementations that support strict notions of atomicity. That is, additional interfaces are needed by an implementation to guarantee that either all consumers receive an event or none of the consumers receive an event; and that all events are received in the same order by all consumers.

4.2 Generic Event Communication

There are two basic models for communicating event data between suppliers and consumers: the *push model* and the *pull model*.

4.2.1 Push Model

In the push model, suppliers “push” event data to consumers; that is, suppliers communicate event data by invoking `push` operations on the *PushConsumer* interface.

To set up a push-style communication, consumers and suppliers exchange *PushConsumer* and *PushSupplier* object references. Event communication can be broken by invoking a `disconnect_push_consumer` operation on the

PushConsumer interface or by invoking a `disconnect_push_supplier` operation on the *PushSupplier* interface. If the *PushSupplier* object reference is nil, the connection cannot be broken via the supplier.

Figure 4-1 illustrates push-style communication between a supplier and a consumer.

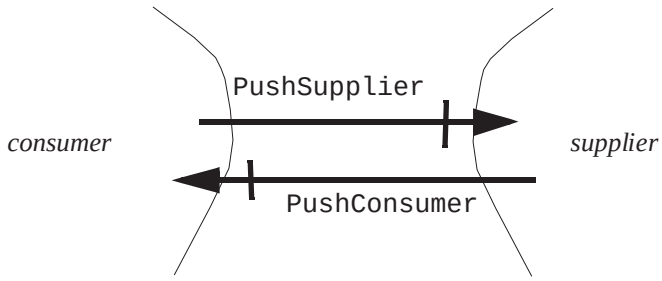

Figure 4-1 Push-style Communication Between a Supplier and a Consumer

4.2.2 Pull Model

In the pull model, consumers "pull" event data from suppliers; that is, consumers request event data by invoking `pull` operations on the *PullSupplier* interface.

To set up a pull-style communication, consumers and suppliers must exchange *PullConsumer* and *PullSupplier* object references. Event communication can be broken by invoking a `disconnect_pull_consumer` operation on the *PullConsumer* interface or by invoking a `disconnect_pull_supplier` operation on the *PullSupplier* interface. If the *PullConsumer* object reference is nil, the connection cannot be broken via the consumer.

Figure 4-2 illustrates pull-style communication between a supplier and a consumer.

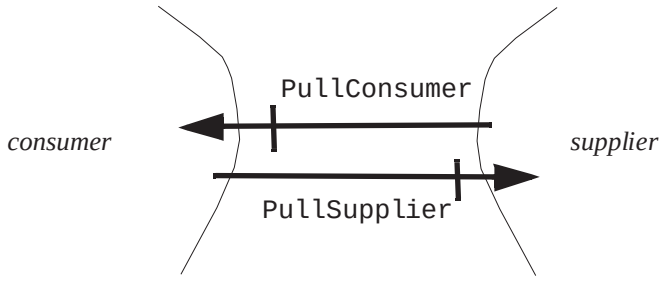

Figure 4-2 Pull-style Communication Between a Supplier and a Consumer

4.3 The CosEventComm Module

The communication styles shown in Figure 4-1 and Figure 4-2 are both supported by four simple interfaces: *PushConsumer*, *PushSupplier*, and *PullSupplier* and *PullConsumer*. These interfaces are defined in an OMG IDL module named `CosEventComm`, as shown in Figure 4-3.

```
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();
    };

};
```

Figure 4-3 The OMG IDL Module CosEventComm

4.3.1 The PushConsumer Interface

A push-style consumer supports the *PushConsumer* interface to receive event data.

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

A supplier communicates event data to the consumer by invoking the `push` operation and passing the event data as a parameter. If the event communication has already been disconnected, the `Disconnected` exception is raised.

The `disconnect_push_consumer` operation terminates the event communication; it releases resources used at the consumer to support the event communication. The *PushConsumer* object reference is disposed.

4.3.2 The PushSupplier Interface

A push-style supplier supports the *PushSupplier* interface.

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

The `disconnect_push_supplier` operation terminates the event communication; it releases resources used at the supplier to support the event communication. The *PushSupplier* object reference is disposed.

4.3.3 The PullSupplier Interface

A pull-style supplier supports the *PullSupplier* interface to transmit event data.

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

A consumer requests event data from the supplier by invoking either the `pull` operation or the `try_pull` operation on the supplier.

- The `pull` operation blocks until the event data is available or an exception is raised.[2] It returns the event data to the consumer. If the event communication has already been disconnected, the `Disconnected` exception is raised.
- The `try_pull` operation does not block: if the event data is available, it returns the event data and sets the `has_event` parameter to *true*; if the event is not available, it sets the `has_event` parameter to *false* and the event data is returned as long with an undefined value. If the event communication has already been disconnected, the `Disconnected` exception is raised.

2. This, of course, may be a standard CORBA exception.

The `disconnect_pull_supplier` operation terminates the event communication; it releases resources used at the supplier to support the event communication. The *PullSupplier* object reference is disposed.

4.3.4 The PullConsumer Interface

A pull-style consumer supports the *PullConsumer* interface.

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

The `disconnect_pull_consumer` operation terminates the event communication; it releases resources used at the consumer to support the event communication. The *PullConsumer* object reference is disposed.

4.4 Event Channels

The *event channel* is a service that decouples the communication between suppliers and consumers. The event channel is itself both a consumer and a supplier of the event data.

An event channel can provide asynchronous communication of event data between suppliers and consumers. Although consumers and suppliers communicate with the event channel using standard CORBA requests, the event channel does not need to supply the event data to its consumer at the same time it consumes the data from its supplier.

4.4.1 Push-Style Communication with an Event Channel

The supplier pushes event data to the event channel; the event channel, in turn, pushes event data to the consumer. Figure 4-4 illustrates a push-style communication between a supplier and the event channel, and a consumer and the event channel.

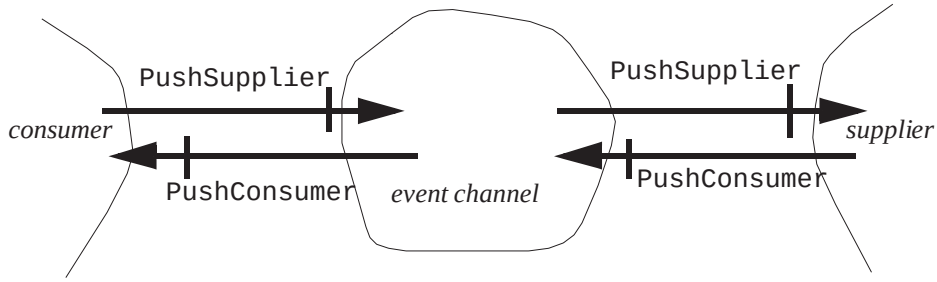

Figure 4-4 Push-style Communication Between a Supplier and an Event Channel, and a Consumer and an Event Channel

4.4.2 Pull-Style Communication with an Event Channel

The consumer pulls event data from the event channel; the event channel, in turn, pulls event data from the supplier. Figure 4-5 illustrates a pull-style communication between a supplier and the event channel, and a consumer and the event channel.

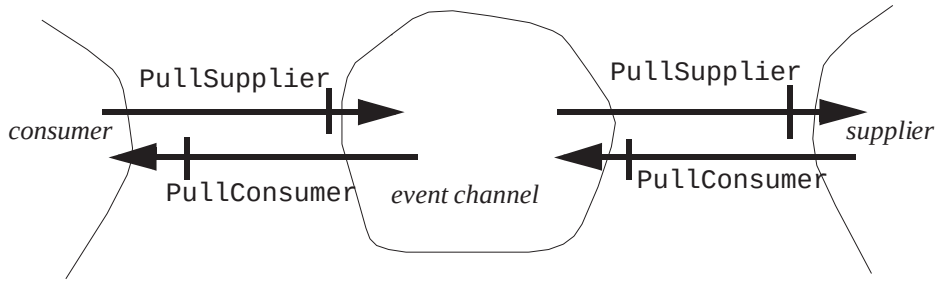

Figure 4-5 Pull-style communication between a supplier and an event channel and a consumer and the event channel

4.4.3 Mixed Style Communication with an Event Channel

An event channel can communicate with a supplier using one style of communication, and communicate with a consumer using a different style of communication.

Figure 4-6 illustrates a push-style communication between a supplier and an event channel, and a pull-style communication between a consumer and the event channel. The consumer pulls the event data that the supplier has pushed to the event channel.

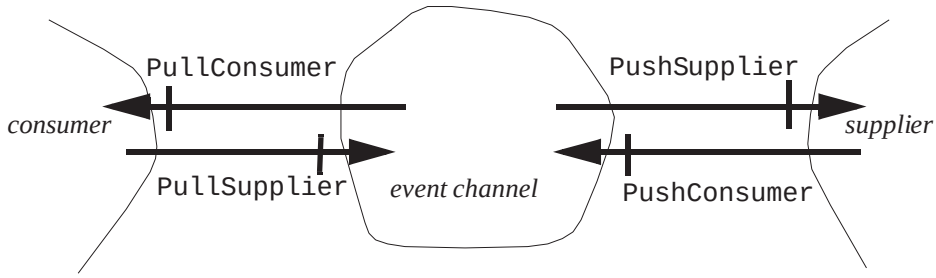

Figure 4-6 Push-style Communication Between a Supplier and an Event Channel, and Pull-style Communication Between a Consumer and an Event Channel

4.4.4 Multiple Consumers and Multiple Suppliers

Figure 4-4, Figure 4-5, and Figure 4-6 illustrate event channels with a single supplier and a single consumer. An event channel can also provide many-to-many communication. The channel consumes events from one or more suppliers, and supplies events to one or more consumers. Subject to the quality of service of a particular implementation, an event channel provides an event to all consumers.

Figure 4-7 illustrates an event channel with multiple push-style consumers and multiple push-style suppliers.

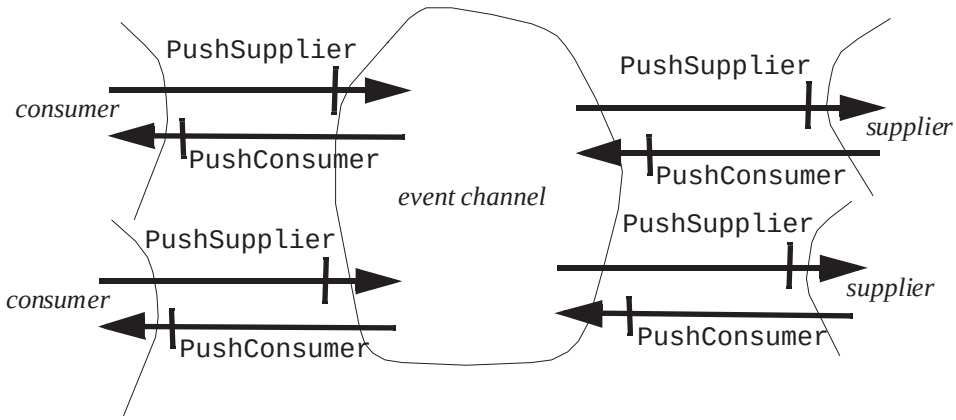

Figure 4-7 An Event Channel with Multiple Suppliers and Multiple Consumers

An event channel can support consumers and suppliers using different communication models.

If an event channel has at least one push-style consumer or at least one pending pull request, the event channel requires an event. If the event channel has pull suppliers, it will issue a request on a pull supplier to satisfy its requirement.

4.4.5 Event Channel Administration

The event channel is built up incrementally. When an event channel is created, no suppliers or consumers are connected to the event channel. Upon creation of the channel, the factory returns an object reference that supports the *EventChannel* interface, as illustrated in Figure 4-8.

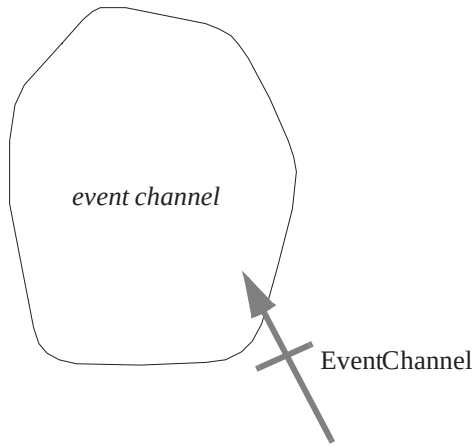

Figure 4-8 A newly created event channel. The channel has no suppliers or consumers.

The *EventChannel* interface defines three administrative operations: an operation returning a *ConsumerAdmin* object for adding consumers, an operation returning a *SupplierAdmin* object for adding suppliers, and an operation for destroying the channel.

The operations for adding consumers return *proxy suppliers*. A proxy supplier is similar to a normal supplier (in fact, it inherits the interface of a supplier), but includes an additional method for connecting a consumer to the proxy supplier.

The operations for adding suppliers return *proxy consumers*. A proxy consumer is similar to a normal consumer (in fact, it inherits the interface of a consumer), but includes an additional method for connecting a supplier to the proxy consumer.

Registration of a producer or consumer is a two step process. An event-generating application first obtains a proxy consumer from a channel, then “connects” to the proxy consumer by providing it with a supplier. Similarly, an event-receiving application first obtains a proxy supplier from a channel, then “connects” to the proxy supplier by providing it with a consumer.

The reason for the two-step registration process is to support composing event channels by an external agent. Such an agent would compose two channels by obtaining a proxy supplier from one and a proxy consumer from the other, and passing each of them a reference to the other as part of their connect operation.

Proxies are in one of three states: *disconnected*, *connected* or *destroyed*. Figure 4-9 gives a state diagram for a proxy. The nodes of the diagram are the states and the edges are labelled with the operations that change the state of the proxy. `Push/pull` operations are only valid in the *connected* state.

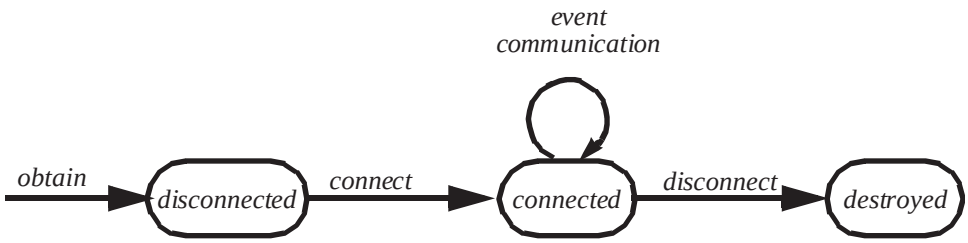

Figure 4-9 State diagram of a proxy.

4.5 The CosEventChannelAdmin Module

The `CosEventChannelAdmin` module defines the interfaces for making connections between suppliers and consumers. The `CosEventChannelAdmin` module is defined in Figure 4-10.

```
#include "CosEventComm.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();
    };

};
```

Figure 4-10 The CosEventChannelAdmin Module

4.5.1 The EventChannel Interface

The *EventChannel* interface defines three administrative operations: adding consumers, adding suppliers, and destroying the channel.

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

Any object that possesses an object reference that supports the *EventChannel* interface can perform these operations:

- The *ConsumerAdmin* interface allows consumers to be connected to the event channel. The `for_consumers` operation returns an object reference that supports the *ConsumerAdmin* interface.
- The *SupplierAdmin* interface allows suppliers to be connected to the event channel. The `for_suppliers` operation returns an object reference that supports the *SupplierAdmin* interface.
- The `destroy` operation destroys the event channel.

Consumer administration and supplier administration are defined as separate objects so that the creator of the channel can control the addition of suppliers and consumers. For example, a creator might wish to be the sole supplier of event data but allow many consumers to be connected to the channel. In such a case, the creator would simply export the *ConsumerAdmin* object.

4.5.2 The ConsumerAdmin Interface

The *ConsumerAdmin* interface defines the first step for connecting consumers to the event channel; clients use it to obtain proxy suppliers.

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

The `obtain_push_supplier` operation returns a *ProxyPushSupplier* object. The *ProxyPushSupplier* object is then used to connect a push-style consumer.

The `obtain_pull_supplier` operation returns a *ProxyPullSupplier* object. The *ProxyPullSupplier* object is then used to connect a pull-style consumer.

4.5.3 The SupplierAdmin Interface

The *SupplierAdmin* interface defines the first step for connecting suppliers to the event channel; clients use it to obtain proxy consumers.

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

The `obtain_push_consumer` operation returns a *ProxyPushConsumer* object. The *ProxyPushConsumer* object is then used to connect a push-style supplier.

The `obtain_pull_consumer` operation returns a *ProxyPullConsumer* object. The *ProxyPullConsumer* object is then used to connect a pull-style supplier.

4.5.4 The ProxyPushConsumer Interface

The *ProxyPushConsumer* interface defines the second step for connecting push suppliers to the event channel.

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

A nil object reference may be passed to the `connect_push_supplier` operation; if so a channel cannot invoke the `disconnect_push_supplier` operation on the supplier; the supplier may be disconnected from the channel without being informed.

If the *ProxyPushConsumer* is already connected to a *PushSupplier*, then the `AlreadyConnected` exception is raised.

4.5.5 The ProxyPullSupplier Interface

The *ProxyPullSupplier* interface defines the second step for connecting pull consumers to the event channel.

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

A nil object reference may be passed to the `connect_pull_consumer` operation; if so a channel cannot invoke a `disconnect_pull_consumer` operation on the consumer; the consumer may be disconnected from the channel without being informed.

If the *ProxyPullSupplier* is already connected to a *PullConsumer*, then the `AlreadyConnected` exception is raised.

4.5.6 The ProxyPullConsumer Interface

The *ProxyPullConsumer* interface defines the second step for connecting pull suppliers to the event channel.

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

Implementations should raise the CORBA standard BAD_PARAM exception if a nil object reference is passed to the `connect_pull_supplier` operation.

If the *ProxyPullConsumer* is already connected to a *PullSupplier*, then the `AlreadyConnected` exception is raised.

An implementation of a *ProxyPullConsumer* may put additional requirements on the interface supported by the pull supplier. If the pull supplier does not meet those requirements the *ProxyPullConsumer* raises the `TypeError` exception. (See section 4.7.2 for an example.)

4.5.7 The ProxyPushSupplier Interface

The *ProxyPushSupplier* interface defines the second step for connecting push consumers to the event channel.

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

Implementations should raise the CORBA standard BAD_PARAM exception if a nil object reference is passed to the `connect_push_consumer` operation.

If the *ProxyPushSupplier* is already connected to a *PushConsumer*, then the `AlreadyConnected` exception is raised.

An implementation of a *ProxyPushSupplier* may put additional requirements on the interface supported by the push consumer. If the push consumer does not meet those requirements the *ProxyPushSupplier* raises the `TypeError` exception. (See section 4.7.1 for an example.)

4.6 Typed Event Communication

Section 4.2 discusses generic event communication using `push` and `pull` operations. The next few sections describe how event communication can be described in OMG IDL and how typed event channels can support such typed event communication.

4.6.1 Typed Push Model

In the typed push model, suppliers call operations on consumers using some mutually agreed interface *I*. The interface *I* is defined in IDL, and may contain any operations subject to the following restrictions:

- All parameters must be `in` parameters only.
- No return values are permitted

These are the same restrictions as CORBA imposes on `oneway` operations, and for similar reasons: event communication is unidirectional, and does not directly support responses. The operations can be declared `oneway`, but need not be.

To set up typed push-style communication, consumers and suppliers exchange *TypedPushConsumer* and *PushSupplier* object references. (Note that the supplier interface is the same as the untyped case.) The supplier then invokes the `get_typed_consumer` operation of the *TypedPushConsumer* interface, which returns an object reference supporting the typed interface, *I*, referred to as an *I-reference*. The particular interface, *I*, that the reference supports is dependent on the

particular *TypedPushConsumer*, and must be mutually agreed by supplier and consumer. Once the supplier has obtained the *I*-reference, it can call operations in interface *I* on the consumer.

As in the case of the generic push-style, event communication can be broken by invoking a `disconnect_push_consumer` operation on the *TypedPushConsumer* interface or by invoking a `disconnect_push_supplier` operation on the *PushSupplier* interface. If the *PushSupplier* object reference is nil, the connection cannot be broken via the supplier.

Figure 4-11 illustrates typed push-style communication between supplier and consumer.

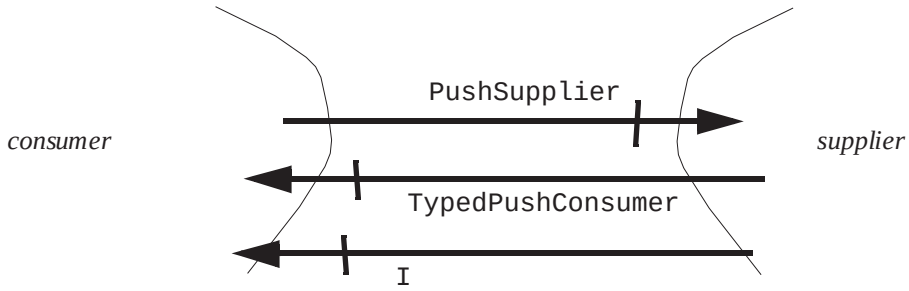

Figure 4-11 Typed Push-style Communication Between a Supplier and a Consumer

4.6.2 Typed Pull Model

In the typed pull model, consumers call operations on suppliers, requesting event information, using some mutually agreed interface *Pull<I>*[3]. For every interface *I* having the properties described in section 4.6.1, an interface *Pull<I>* is defined as follows:

- For every operation `o` in *I*, *Pull<I>* contains two operations:
  - `pull_o`, with all `in` parameters changed to `out` parameters. When called, this operation will return with the event data in the *out* parameters. If no *o*-event is currently available, it will block.
  - `boolean try_o`, with all `in` parameters changed to out parameters. When called, this operation will check whether an *o*-event is currently available. If so, it will return `true`, with the event data in the *out* parameters. If not, it will return `false`, with the `out` parameters undefined

3. *Pull<I>* is used as notation for a computed interface from interface I. Thus, if *I* is an interface *DocumentEvents*, *Pull<I>* is an interface *PullDocumentEvents*.

The interface *Pull<I>* is designed to allow pulling of exactly the same events that can be pushed using interface *I*.

To set up typed pull-style communication, consumers and suppliers exchange *PullConsumer* and *TypedPullSupplier* object references. (Note that the consumer interface is the same as the untyped case.) The consumer then invokes the `get_typed_supplier` operation of the *TypedPullSupplier*, which returns an object reference supporting the typed interface, *Pull<I>*, referred to as a *Pull<I>-reference*. The particular interface, *Pull<I>*, that the reference supports is dependent on the particular *TypedPullSupplier*, and must be mutually agreed by supplier and consumer. Once the consumer has obtained the *Pull<I>-reference*, it can call operations in interface *Pull<I>* on the supplier.

Figure 4-12 illustrates typed pull-style communication between supplier and consumer.

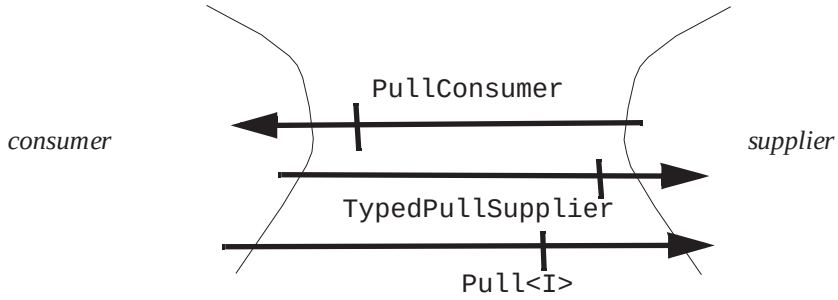

Figure 4-12 Typed Pull-style Communication Between a Supplier and a Consumer

4.7 The CosTypedEventComm Module

The typed communication styles shown in Figure 4-11 and Figure 4-12 are both supported by two new interfaces, *TypedPushConsumer* and *TypedPullSupplier* and two existing interfaces, *PushSupplier* and *PullConsumer*. The first two interfaces are

defined in an OMG IDL module named *CosTypedEventComm*, as shown in Figure 4-13. The last two are the same as for untyped event communication, and were defined in the *CosEventComm* module in Figure 4-3.

```
#include "CosEventComm.idl"

module CosTypedEventComm {

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

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

};
```

Figure 4-13 The IDL Module `CosTypedEventComm`

4.7.1 The TypedPushConsumer Interface

A typed push-style consumer supports the *TypedPushConsumer* interface both to receive event data in the generic manner, and to supply a specific typed interface through which to receive it in typed form.

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

The *TypedPushConsumer* can behave just like an untyped *PushConsumer*, described in section 4.3.1. In addition, if the supplier wishes to communicate event data to the consumer in typed rather than generic form, it first invokes the `get_typed_consumer` operation. This returns an *I-reference* supporting an interface *I*. The particular interface, *I*, that the reference supports is dependent on the particular *TypedPushConsumer*. The return type of the operation is *Object*, because different *TypedPushConsumers* will return references of different types, so the actual type cannot be specified in a general definition. Once the supplier has obtained the *I-reference*, it can narrow it to *I*, and then call operations in interface *I* on the consumer. Mutual agreement about *I* is needed between the supplier and consumer. If they do not agree, the narrow operation will fail.

As noted above, a *TypedPushConsumer* must support the `push` operation, inherited from Cos*EventComm::PushConsumer*. Implementing `push` fully is an unnecessary burden if the consumer is intended for typed use only. It is therefore permissible to implement a *TypedPushConsumer* with a null implementation of `push` that merely raises the standard CORBA exception `NO_IMPLEMENT`. Clearly, suppliers must know this and confine themselves to typed communication with such consumers.

4.7.2 The TypedPullSupplier Interface

A typed pull-style supplier supports the *TypedPullSupplier* interface both to allow consumers to pull event data in the generic manner, and to supply a specific typed interface through which they can pull it in typed form.

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

The *TypedPullSupplier* can behave just like an untyped *PullSupplier*, described in section 4.3.3. In addition, if the consumer wishes to pull event data from the supplier in typed rather than generic form, it first invokes the `get_typed_supplier` operation. This returns a *Pull<I>-reference* supporting an interface *Pull<I>*. The particular interface, *Pull<I>*, that the reference supports is dependent on the particular *TypedPullSupplier*. The return type of the operation is *Object*, because different *TypedPullSuppliers* will return references of different types, so the actual type cannot be specified in a general definition. Once the consumer has obtained the *Pull<I>-reference*, it can narrow it to *Pull<I>*, and then call operations in interface *Pull<I>* on the supplier. Mutual agreement about *Pull<I>* is needed between the supplier and consumer. If they do not agree, the `narrow` operation will fail.

As noted above, a *TypedPullSupplier* must support the `pull` and `try_pull` operations, inherited from Cos*EventComm::PullSupplier*. Implementing these operations fully is an unnecessary burden if the supplier is intended for typed use only. It is therefore permissible to implement a *TypedPullSupplier* with null implementations of `pull` and `try_pull` that merely raise the standard CORBA exception `NO_IMPLEMENT`. Clearly, consumers must know this and confine themselves to typed communication with such suppliers.

4.8 Typed Event Channels

Typed event channels are analogous to generic event channels, but they support both typed and generic event communication. These forms can be mixed at will. A single channel can handle events supplied and consumed in any combination of the forms defined earlier (push/pull, generic/typed). An event supplied in typed form can be consumed in generic form, or vice versa.[4]

4. Doing this does require an understanding on the part of the generic suppliers and consumers of how the channel packages parameters of typed calls when converting them to generic form. Details of this packaging are dependent on the implementation of the channel.

4.9 The CosTypedEventChannelAdmin Module

The `CosTypedEventChannelAdmin` module defines the interfaces for making connections between suppliers and consumers that use either generic or typed communication. It is defined in Figure 4-14. Most of its interfaces are specializations of the corresponding interfaces in the `CosEventChannel` module defined in Figure 4-10.

```
#include “CosEventChannel.idl”
#include “CosTypedEventComm.idl”

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

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

Figure 4-14 The CosTypedEventChannelAdmin Module

4.9.1 The TypedEventChannel Interface

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

This interface is analogous to Cos`EventChannelAdmin::EventChannel`. However, it returns typed versions of the consumer and supplier administration interfaces, which are capable of providing proxies for either generic or typed communication.

4.9.2 The TypedConsumerAdmin Interface

The *TypedConsumerAdmin* interface defines the first step for connecting consumers to typed event channel; clients use it to obtain proxy suppliers.

```
interface TypedConsumerAdmin :
        CosEventChannelAdmin::ConsumerAdmin {
    TypedProxyPullSupplier obtain_typed_pull_supplier(
            in Key supported_interface)
        raises (InterfaceNotSupported);
    ProxyPushSupplier obtain_typed_push_supplier(
            in Key uses_interface)
        raises(NoSuchImplementation);
};
```

The `obtain_typed_pull_supplier` operation takes a Key parameter that identifies an interface, *Pull<I>*. The scope of the key is the typed event channel. It returns a *TypedProxyPullSupplier* for interface *Pull<I>*. The *TypedProxyPullSupplier* will allow an attached pull consumer to pull events either in generic form or using operations in interface *Pull<I>*. It is up to the implementation of `obtain_typed_pull_supplier` to create or find an appropriate *TypedProxyPullSupplier*. If it cannot, it raises the exception `InterfaceNotSupported`.

The `obtain_typed_push_supplier` operation takes a `Key` parameter that identifies an interface, *I*. The scope of the key is the typed event channel. It returns a *ProxyPushSupplier* that calls operations in interface *I*, rather than `push` operations. It is up to the implementation of `obtain_typed_push_supplier` to create or find an appropriate *ProxyPushSupplier*[5]. If it cannot, it raises the exception `NoSuchImplementation`.

Such a *ProxyPushSupplier* is guaranteed only to invoke operations defined in interface *I*. Any event on the channel that does not correspond to an operation defined in interface *I* is not passed on to the consumer. Such a *ProxyPushSupplier* is therefore an event filter based on type.

4.9.3 The TypedSupplierAdmin Interface

The *TypedSupplierAdmin* interface defines the first step for connecting suppliers to the typed event channel; clients use it to obtain proxy consumers.

```
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 `obtain_typed_push_consumer` operation takes a `Key` parameter that identifies an interface, *I*. The scope of the key is the typed event channel. It returns a *TypedProxyPushConsumer* for *I*. An attached supplier can provide events by using operations in interface *I*. It is up to the implementation of `obtain_typed_push_consumer` to create or find an appropriate *TypedProxyPushConsumer*. If it cannot, it raises the exception `InterfaceNotSupported`.

The `obtain_typed_pull_consumer` operation takes a `Key` parameter that identifies an interface, *Pull<I>*. The scope of the key is the typed event channel. It returns a *ProxyPullConsumer* that calls operations in interface *Pull<I>*, rather than `pull` operations. It is up to the implementation of `obtain_typed_pull_consumer` to create or find an appropriate *ProxyPullConsumer*. If it cannot, it raises the exception `NoSuchImplementation`.

Such a *ProxyPullConsumer* is guaranteed only to invoke operations defined in interface *Pull<I>*. Any event request that does not correspond to an operation defined in interface *Pull<I>* is not pulled from the supplier. Such a *ProxyPullConsumer* is therefore an event filter based on type.

5. see Appendix A for implementation considerations.

4.9.4 The TypedProxyPushConsumer Interface

The *TypedProxyPushConsumer* interface defines the second step for connecting push suppliers to the typed event channel.

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

- By inheriting from both Cos`EventChannelAdmin::ProxyPushConsumer` and Cos`TypedEventComm::TypedPushConsumer`, this interface supports:
- Connection and disconnection of push suppliers, exactly as in the generic event channel,
- Generic `push` operation and
- Obtaining the typed view, so that the supplier can use typed push communication. The reference returned by `get_typed_consumer` has the interface identified by the `Key` used when this *TypedProxyPushConsumer* was obtained. (See section 4.9.3)

4.9.5 The TypedProxyPullSupplier Interface

The *TypedProxyPullSupplier* interface defines the second step for connecting pull consumers to the typed event channel.

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

By inheriting from both Cos`EventChannelAdmin::ProxyPullSupplier` and Cos`TypedEventComm::TypedPullSupplier`, this interface supports:

- Connection and disconnection of pull consumers, exactly as in the generic event channel,
- Generic `pull` and `try_pull` operations and
- Obtaining the typed view, so that the consumer can use typed pull communication. The reference returned by `get_typed_supplier` supports the interface identified by the `Key` used when this *TypedProxyPullSupplier* was obtained. (See section 4.9.2).

4.10 Composing Event Channels and Filtering

The event channel administration operations defined in section 4.5 support the composition of event channels. That is, one event channel can consume events supplied by another. This architecture allows the implementation of an event channel that filters the events supplied by another.

Since the *ProxyPushSupplier* for interface *I* of a typed event channel only pushes events that correspond to *I*, it acts as a filter based on type. Similarly, the *ProxyPullConsumer* for interface *Pull<I>* of a typed event channel only pulls events that correspond to *Pull<I>*, it also acts as a filter based on type.

4.11 Policies for Finding Event Channels

The Event Service does not establish a policy for finding event channels. Finding a service is orthogonal to using the service. Higher levels of software (such as the desktop) can make policies for using the event channel. That is, higher layers will dictate when an event channel is created and how references to the event channel are obtained. By representing the event channel as an object, it has all of the properties that apply to objects, including support by finding mechanisms.

For example, when a user performs a `drag-and-drop` or `cut-and-paste` operation, an event channel could be created and identified to suppliers and consumers. Alternatively, the event channel could be named in a naming context, or it could be exported through an operation on an object.

Appendix A Implementing Typed Event Channels

Note – Implementation details do not form part of an OMG specification, and should not be standardized. On the other hand, it is not obvious that typed channels can be implemented without extensions to CORBA. This section indicates *one* strategy for implementing typed event channels. It is included to show that typed event channels can be implemented; it is not intended in any way to constrain implementations. Optimized implementations are certainly possible.

Figure 4-15 demonstrates a possible implementation of a typed event channel. This appendix concentrates on push style communication. The implementation of pull-style communication is analogous.

The implementation interposes an *encoder* between typed-style suppliers and the channel and a *decoder* between the channel and typed-style consumers.

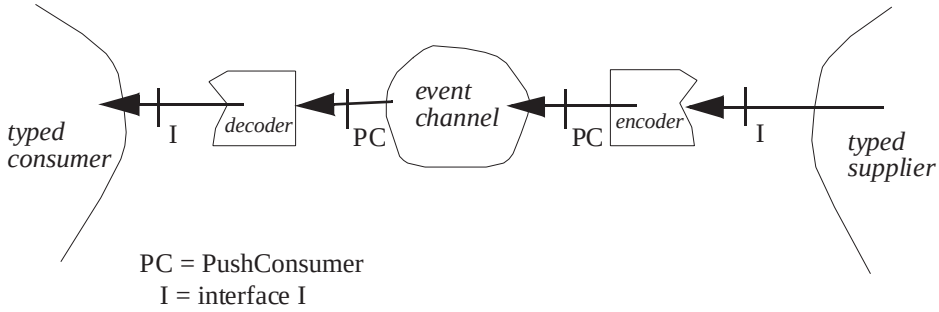

Figure 4-15 A possible implementation of a typed event channel.

At the supplier end, an *encoder* converts operation calls to `push` calls.

At the consumer end, a *decoder* converts `push` calls back to operation calls.

The effect of such a communication is thus that the original operation is eventually called on the consumer, but the communication is routed via the channel. Of course, there can be multiple suppliers and multiple consumers on the same channel. Whenever one of the suppliers calls an operation, it is delivered by the channel to all consumers.

The encoder must package the operation identification and the parameters in a manner that the decoder can unpack them correctly.

Given the OMG IDL definition of an interface, *I*, an encoder generator could generate an implementation that supports the interface *I* and converts all calls on this interface to `push` calls on an event channel.

Similarly, it is possible to generate an I-decoder from the OMG IDL definition of I.

The typed event channel is responsible for finding, creating or implementing the appropriate encoders. An appropriate encoder is found or created in response to the `obtain_typed_push_consumer` request on the typed event channel. The encoder is returned in response to the `get_typed_consumer` request.

Similarly, the typed event channel is responsible for finding, creating or implementing the appropriate decoders. An appropriate decoder is found or created in response to the `connect_push_consumer` request on the typed event channel.

Appendix B An Event Channel Use Example

This section illustrates an example use of the event channel, including the following:

- Creating an event channel
- Consumers and/or suppliers finding the channel
- Suppliers using the event channel
- In this example, the document object creates event channels and defines operations in its interface to allow consumers to be added.
- The *Document* interface defines two operations to return event channels:

```
interface Document {

    ConsumerAdmin title_changed();

    ConsumerAdmin new_section();

        :

};
```

The `title_changed` operation causes the document to generate an event when its title is changed; the `new_section` operation causes the document to generate an event when a new section is added. Both operations return *ConsumerAdmin* object references. This allows consumers to be added to the event channel.

- The `title_changed` implementation contains instance variables for using and administering the event channels.

```
/* Factory for creating event channels. */
EventChannelFactoryRef    ecf;

/* For title changed event channel */
EventChannelRef                 event_channel;

ConsumerAdminRef              consum_admin;
SupplierAdminRef         supplier_admin;

ProxyPushConsumerRef      proxy_push_consumer;
PushSupplierRef           doc_side_connection;
```

- At some point, the document implementation creates the event channel, gets supplier and consumer administrative references, and adds itself as a supplier[6].

```
event_channel = ecf->create_eventchannel(env);

supplier_admin = event_channel->for_suppliers(env);
consumer_admin = event_channel->for_consumers(env);
proxy_push_consumer = supplier_admin->obtain_push_consumer(env);

proxy_push_consumer->connect_push_supplier(env,
                                           doc_side_connection)
```

- The `title_changed` operation returns the *ConsumerAdmin* object reference.

```
return consumer_admin;
```

  Clients of this operation can add consumers.

- When the title changes, the document implementation pushes the event to the channel.

```
proxy_push_consumer->push(env,data);
```

The document implementation similarly initializes, exports, and uses the event channel for reporting new sections.

6. For readability, exception handling is omitted from these code fragments.