Oracle® Tuxedo

Using the CORBA Notification Service

11*g* Release 1 (11.1.1.1.0)

March 2010

ORACLE®

Oracle Tuxedo Using the CORBA Notification Service, 11*g* Release 1 (11.1.1.1.0)

Contents

5. Building the Introductory Sample Application

6. Building the Advanced Sample Application

7. CORBA Notification Service Administration

Index

CHAPTER 1

Overview

This topic includes the following sections:

Introduction

The Notification Service provides an event service for the Oracle Tuxedo CORBA environment. It is not meant to be a standalone product, but rather a layered product on Oracle Tuxedo.

The Notification Service offers similar capabilities to those of the Oracle Tuxedo EventBroker, but with a programming model and interface that is natural for CORBA users. A side effect of this approach is that the majority of the CORBA-based Notification Service is not supported since it is either incompatible with, or provides capabilities well beyond that of the Oracle Tuxedo EventBroker.

The Notification Service is an Oracle Tuxedo subsystem that receives event posting messages, filters them, and distributes them to subscribers. A poster is an Oracle Tuxedo CORBA application that detects when an event of interest has occurred and reports (posts) it to the Notification Service. A subscriber is an Oracle Tuxedo CORBA application that requests that some notification action be taken when an event of interest is posted.

The concept of an “anonymous” service—the Notification Service—that receives and distributes messages provides another client-server communication paradigm to Oracle Tuxedo CORBA

environment. Instead of a one-to-one relationship between a requester and a provider, an arbitrary number of posters can post a message for an arbitrary number of subscribers. The posters simply post events, without knowing who receives the information or what is done about it. The subscribers can receive whatever information they are interested in from the Notification Service, without knowing who posted it, and subscribers can be notified and take action in a variety of ways.

Typically, Notification Service applications are designed to handle exception events. The application designer has to decide what events in the application need to be monitored. In a banking application, for example, an event might be posted for an unusually large withdrawal transaction; but it would not be particularly useful to post an event for every withdrawal transaction. And not all users would need to subscribe to that event; perhaps just the branch manager, would need to be notified.

The programming model for the Notification Service is based on the CORBA programming model. There are two sets of interfaces: one is a minimal subset of the CORBA-based Notification Service interface (referred to in this document as the CosNotification Service interface), and the other is the Oracle Simple Events interface (an Oracle proprietary interface) designed to be easy to use. Both interfaces pass standard, structured events, as defined by the CORBA-based Notification Service specification.

The two interfaces are compatible with each other; that is, events posted using the CosNotification Service interface can be subscribed to by the Oracle Simple Events interface and vice versa.

Functional Overview

The Notification Service system comprises three basic components (see Figure 1-1):

- The event poster, or supplier.

  The supplier is the producer of events. It creates events and posts them to the Notification Service.

- The Notification Service, also known as the event channel.

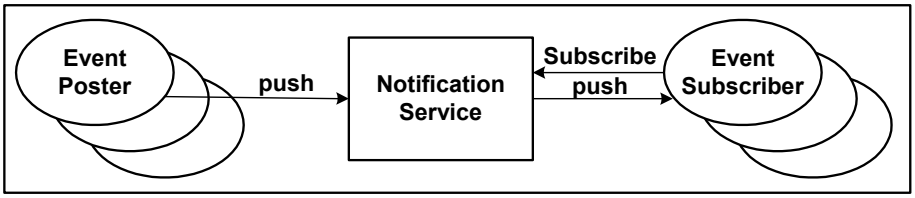
  The Notification Service processes events.

- The event subscriber, or consumer.

  The consumer is the recipient of the events. It connects to the Notification Service and subscribes to some set of events.

When the Notification Service receives an event that matches a consumer's subscription, it attempts to deliver the event to that consumer. There can be many suppliers and consumers. Logically, there is only one Notification Service, even though the Notification Service can be replicated.

Figure 1-1 Notification Service Model



According to the CORBA-based Notification Service specification, event posters always use the push model. Thus, event posters push events to the Notification Service by invoking an operation. The Notification Service takes responsibility for filtering and delivering the event. There is no direct association between event posters and event subscribers. At any point in time there may be zero, one, or many event posters or event subscribers.

Also, according to the CORBA-based Notification Service specification, subscribers can select one of two event delivery models, push or pull. Only the push model is supported in this release of Oracle Tuxedo. Thus, the Notification Service pushes events to the consumer by invoking an operation on the consumer. Depending on the Quality of Service (QoS) of the matching subscription, the event might be stored durably, pending delivery to the consumer.

Product Components

The Oracle Tuxedo CORBA Notification Service supports the following:

- An Oracle Simple Events application programming interface (API) for ease-of-use.
- A minimal set of operations defined by the CosNotification Service API.
- Two Qualities of Service (QoS) for subscriptions: transient and persistent.

  For transient subscriptions, the Notification Service makes only one attempt to deliver the event to a subscriber. If that attempt fails, the event is discarded and if the Notification Service determines that the subscriber is shutdown or otherwise not available, the subscription is cancelled.

For persistent subscriptions, if the first delivery attempt fails, the Notification Service holds the event and keeps attempting to deliver the subscription until the configurable retry limit is reached. After the retry limit is reached, the Notification Service moves the event to an error queue, where it is held for disposition by the system administrator. The system administrator either removes the event from the error queue, which in effect discards it, or moves it back to the pending queue so that further attempts to deliver it can be made.

- Using the `UBBCONFIG` file for initial configuration of the system, event queues, and server processes.
- Using the Oracle Tuxedo style FML field tables. Through the use of FML field tables, the Notification Service can support:
  - Event data filtering between event posters and event subscribers.
  - Interoperability with Oracle Tuxedo EventBroker such that events posted by the Notification Service can be consumed by the Tuxedo EventBroker and vice versa.
- Using the following Oracle Tuxedo Notification Service servers to process events:
  - TMNTS
  - TMNTSFWD_P
  - TMNTSFWD_T
- Using the following Oracle Tuxedo system servers to process events:
  - TMSYSEVT
  - TMUSREVT
  - TMQUEUE
  - TMQFORWARD
- Using the Oracle Tuxedo `ntsadmin` administrative utility to manage event queues.
- Using the Oracle Tuxedo `qmadmin` administrative utility to configure and manage event queues.
- Using the Oracle Tuxedo `tmadmin` administrative utility to configure and manage transaction logs.

CORBA Notification Service API Reference

This topic includes the following sections:

Note: The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Introduction

The Oracle Tuxedo CORBA Notification Service supports two application programming interfaces. One is based on the CORBA-based Notification Service as defined by the *CORBAservices: Common Object Services Specification*. This interface is referred to in this document as the CosNotification Service interface. The other interface, called the Oracle Simple Events interface, is an Oracle proprietary interface designed as an easier to use alternative.

Both interfaces pass structured events as defined by the CORBA-based Notification Service specification and are compatible with each other; that is, events posted using the CosNotification Service interface can be subscribed to by the Oracle Simple Events interface and vice versa.

Before using the Notification Service APIs, consider the following topics:

- Quality of Service
- Obtaining the Channel Factory
- Using Transactions
- Structured Event Fields, Types, and Filters
- Creating FML Field Table Files for Events
- Interoperability with Oracle Tuxedo Applications

Quality of Service

To determine the persistence of the subscription and whether or not events delivery is retried following a failed delivery, subscribers specify a Quality of Service (QoS). There are two Quality of Service settings: *persistent* and *transient* Quality of Service (QoS). The QoS is a property of the subscription.

Persistent Subscriptions

Persistent subscriptions provide strong guarantees about event delivery and the permanence of the subscription. Persistent subscriptions do come with a cost, however, as they consume more system resources (for example, disk space, CPU cycles, and so on), and require more administration (such as managing queues and detecting dead subscribers).

Persistent subscriptions exhibit the following properties:

- The subscription is in effect until an unsubscribe operation is performed. This means that a subscriber application can be shut down and its subscription can still be active. In this case, events are stored for the subscriber and, when the subscriber restarts, are delivered to the subscriber without it having to recreate the subscription.
- If an event cannot be delivered, event delivery is retried until the administrative retry limit is exceeded.When the event retry limit has been exceeded, the event is moved from the pending queue to an error queue. An administrator can move events from the error queue back to the pending queue, where delivery attempts will restart.

- If an event is successfully delivered to a subscriber, but the Notification Service for some reason does not receive the “successful delivery” return message, the Notification Service may deliver the same event more than once.

Transient Subscriptions

Transient subscriptions provide the best performance with the least overhead and exhibit the following properties:

- One attempt is made to deliver the event to each matching subscription. If that attempt fails, the event is lost.

The subscription is in effect until a failed event delivery is detected. On detection of a failed delivery, the subscription is terminated. Normally, the Notification Service, for performance reasons, does not check whether it successfully delivered an event to a transient subscriber. However, occasionally, when the Notification Service delivers an event to a transient subscriber, it checks whether or not the event was successfully delivered. If it was not successfully delivered and the `CORBA::TRANSIENT` exception is not returned, the Notification Service assumes that the subscription has gone away and cancels the subscription. If the Notification Service receives the `CORBA::TRANSIENT` exception when an attempt to deliver fails, it assumes that the subscriber is busy and discards the event, but it does not cancel the subscription.

The automatic cancellation of dead transient subscriptions provides a cleanup mechanism for transient subscribers that forget to unsubscribe. Note, however, that the Notification Service checks for successful delivery the first time it sends an event to a subscriber, but does not perform it again until five minutes have elapsed and it delivers another event. Therefore, the interval between checks is at least five minutes, but will be longer if there is no event to deliver when five minutes have elapsed. The minimum interval of five minutes is fixed and cannot be changed. Therefore, event delivery failure is not necessarily detected on the first failed delivery attempt. It is only detected when the Notification Service checks.

Obtaining the Channel Factory

The Channel Factory is used by event poster applications and subscriber applications to find the event channel. The event channel is then used to post events and to subscribe, or create subscriptions, and unsubscribe, or cancel subscriptions.

Notification Service applications use the Bootstrap object to obtain an object reference to the event channel factory. This is done by using the `Tobj_Bootstrap::resolve_initial_references` operation. The Bootstrap object supports two service IDs for Notification Service applications, `NotificationService` and

Tobj_SimpleEventsService. The NotificationService object is used in applications that use the CosNotification Service API. The Tobj_SimpleEventsService object is used in applications that use the Oracle SimpleEvents API.

Service ID	Object Type
NotificationService	CosNotifyChannelAdmin::EventChannelFactory
Tobj_SimpleEventsService	Tobj_SimpleEvents::ChannelFactory

Note: Release 8.0 of Oracle Tuxedo CORBA continues to include the Oracle client environmental objects provided in previous releases of Oracle WebLogic Enterprise for use with the Tuxedo 8.0 CORBA clients. Oracle Tuxedo 8.0 clients should continue to use these environmental objects to resolve initial references bootstrapping, security and transaction objects. In release 8.0 of Oracle Tuxedo CORBA, support has been added for using the OMG Interoperable Naming Service (INS) to resolve initial references to bootstrapping, security, and transaction objects. For information on INS, see the *CORBA Programming Reference*.

Using Transactions

The behavior regarding transactions is the same for the Oracle SimpleEvents API and the CosNotification Service API. The only operation that supports transactional behavior is push_structured_event, which is supported by the CosNotifyChannelAdmin::StructuredProxyPushConsumer and Tobj_SimpleEvents::Channel interfaces. All other operations can be used in the context of a transaction, but work the same regardless of whether they are executed in a transaction or not.

The behavior when posting an event is tied to the QoS of the subscription. If an event is posted in the context of a transaction, and the event delivery QoS of the subscription is persistent, the delivery will be affected by the outcome of the transaction; that is, if the transaction is committed, the Notification Service attempts to deliver the event to subscribers as it normally would. If the transaction is rolled back, then the Notification Service does not attempt to deliver the event.

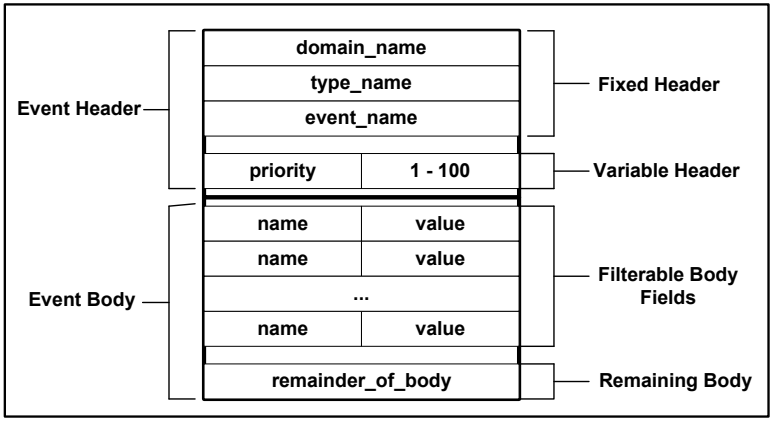
If an event is posted in the context of a transaction, and the event delivery QoS of the subscriber's subscription is transient, one attempt will be made to deliver the event, regardless of the transaction outcome. That is, the transaction has no effect on whether the event is delivered or not, and one attempt will be made to deliver the event.

Note: There is no transaction context associated with event delivery. However, in the case of persistent subscriptions, once the poster's transaction commits, the Notification Service guarantees that the event will be delivered to the subscriber or put on the error queue to await administrative action.

Structured Event Fields, Types, and Filters

All events that are either pushed by posters to the Notification Service, or delivered to subscribers, are COS Structured Events; that is, they conform to the definition of Structured Events as specified by the CORBA-based Notification Service—a service which extends the CORBAservices Event Service (see Figure 2-1). If the events are to be filtered based on content (versus filtering on domain and type), or if the events are going to be subscribed to by Oracle Tuxedo applications, then additional restrictions apply. The restrictions apply to data types and filtering based on event content. These restrictions are explained below.

Figure 2-1 Structured Event



- The Fixed Header section consists of three fields that can be used when you create structured events: fixed_header.event_type.domain_name and fixed_header.event_type.type_name, and fixed_header.event_type.event_name. When an event is posted all three of the these fields are passed in the Notification Service. However, when subscriptions are created, only the first two fields, domain_name and type_name, are used to filter events. These fields are defined in the subscription as regular expressions. The event_name field cannot be used in subscriptions.
- The Variable Header consists of a single name/value (NV) pair, namely Priority. Priority can take a value in the range 1-100 (versus a range of –32767 to 32767 as specified in

CORBA Notification Service specification). Priority is used internally to the system to prioritize the processing of events. The highest priority is 100. There is no guarantee that higher priority events will, in fact, be given priority over lower priority events. The support provided for the Variable Header differs from that specified in the CORBA Notification Service specification in two ways: first, there is a single field supported (Priority) versus the five fields listed in the specification; and second, user-defined fields are supported, but no action is taken in response to their content. The user-defined fields are merely passed through.

- The Filterable Body consists of zero or more NV pairs. The values in these pairs are limited to the following types: `any`, `long`, `unsigned long`, `short`, `unsigned short`, octet, `char`, `float`, `double`, `string`, `boolean`, `void`, and `null`. These fields can be used in filter expressions.
- The Remaining Body consists of a single `ANY`. The value is limited to the following types: `any`, `long`, `unsigned long`, `short`, `unsigned short`, octet, `char`, `float`, `double`, `string`, `boolean`, `void`, and `null`. This field cannot be used in a filter expression.

Designing Events

The design of events is basic to any notification service. The design impacts not only the volume of information that is delivered to matching subscriptions, but the efficiency and performance of the Notification Service as well. Therefore, careful planning should be done to ensure that your Notification Service will be able to handle your needs now and allow for future growth.

The Notification Service supports five levels of event design: (1) domain name, (2) type name, (3) priority, (4) filterable data, and (5) remainder of body. When designing an event, you must specify a domain name and a type name; priority and filterable data are optional. The domain name you choose can relate to your business. Hospitals, for example, are in the health care business, so for a Notification Service application for a hospital you might choose "HEALTHCARE" as a domain name. You might want to categorize the events by the type of insurance provider, so you may choose "HMO" or "UNINSURED" as the type name. You may want to further define the events by the entity responsible for payment, so you might choose to use the filterable data to identify the entity as "billing" for a specific "HMO_Account" or a specific or "Patient_Account." Listing 2-1 shows an example of this type of event design.

Listing 2-1 Event Design

```
domain_name = "HEALTHCARE"
type_name = "HMO"
```

```
#Filterable data name/value pairs.
filterable_data.name = "billing"
filterable_data.value = 4498
filterable_data.name = "patient_account"
filterable_data.value = 37621
```

Obviously, the more specific and precise you are in designing the events that you want your Notification Service application to post and receive, the fewer will be the events the Notification Service will have to process. This has a direct impact on system resources and configuration requirements. Therefore, a lot of thought should be given to event design.

Creating FML Field Table Files for Events

You must create Field Manipulation Language (FML) field table files for events only if one of the following capabilities is required; otherwise FML tables are not required.

- Event data filtering (in addition to domain and type fields) between Oracle Tuxedo event posters and subscribers
- Interoperability between the Oracle Tuxedo Notification Service and the Oracle Tuxedo EventBroker

A structured event's `filterable_data` field contains a list of name/value (NV) pairs. An event's data is typically stored in this list. The field names in the FML field table files must match the name in the structured event. The field type can be any allowable FML type (`long`, `short`, `double`, `float`, `char`, `string`) except `carray`. The value in the structured event must be the same type as defined in the field table. Table 2-1 shows the CORBA Any Types supported by Oracle Tuxedo, and which ones can be used for data filtering and Oracle Tuxedo interoperability.

Table 2-1 Supported CORBA Any Types

CORBA Any Types	Supported for Data Filtering and Tuxedo Interoperability
`short`	Yes
`long`	Yes
`unsigned short`	No

Table 2-1 Supported CORBA Any Types (Continued)

CORBA Any Types	Supported for Data Filtering and Tuxedo Interoperability
`unsigned long`	No
`float`	Yes
`double`	Yes
`char`	Yes
`boolean`	No
`octet`	No
`string`	Yes
`void`	No
`null`	No
`any`	No

Listing 2-2 shows an example of an FML field table file. The `*base 2000` is the base number for the fields. The first entry has a field name of `billing`, a field number of `1` relative to the base, and a field type of `long`.

Listing 2-2 Data Filtering FML Field Table File

```
*base 2000
#Field Name       Field #  Field Type    Flags     Comments
#-----------      -------  ----------    ------    --------
billing           1        long            -         -
stock_name        2        string          -         -
price_per_share   3        double          -         -
number_of_shares  5        long            -         -
```

The following guidelines and restrictions apply to Oracle Tuxedo FML field table files:

- The FML filename cannot exceed 15 characters in length.
- Because Oracle Tuxedo uses FML32, the base number plus the field number is restricted to be between 101 and 33,554,431, inclusive.
- When FML is used with other software that also uses fields, additional restrictions may be imposed on field numbers.

For information on how to create and configure FML field table files, see `field_tables` in the *Oracle Tuxedo Command Reference* and the *Programming Oracle Tuxedo ATMI Applications Using FML*.

Interoperability with Oracle Tuxedo Applications

Applications that use the Oracle Tuxedo CORBA Notification Service are interoperable with Oracle Tuxedo applications that use the Oracle Tuxedo EventBroker. An application using the Oracle Tuxedo Notification Service can post events that are delivered to Oracle Tuxedo EventBroker subscribers, and can receive events that have been posted by Oracle Tuxedo EventBroker.

To achieve this interoperability, it is necessary to understand the mapping between CosNotification Structured Events and the Oracle Tuxedo FML buffer so that the contents of the FML field tables can be coordinated by Oracle Tuxedo. There are two cases to consider: posting events that are to be received by Oracle Tuxedo applications via Oracle Tuxedo EventBroker; and receiving events that have been posted to the Notification Service Event Channel by Oracle Tuxedo applications.

Posting Events

For an Oracle Tuxedo application to subscribe to events posted by an Oracle Tuxedo application, you must understand how an Oracle Tuxedo structured event is mapped to FML32 and the event name at posting time. The mapping is as follows:

- The domain_name and type_name are assembled into a string in the form `domain_name.type_name` to form the event name. This is the event name (eventname parameter) used on the `tppost` operation.
- Each name/value (NV) pair in the Filterable Body and the variable header portion of the structured event is mapped to an FML32 field of the same name if the field is also defined in FML. If you set the domain to `"TMEVT"`, then the event name equals the type name.

Receiving Events

Oracle Tuxedo system events and user events can be received by Oracle Tuxedo applications. System events are generated by the Oracle Tuxedo system—not by applications. User events are generated by Oracle Tuxedo applications. For a listing of System events see `EVENTS` in the *Oracle Tuxedo Command Reference*. System events and user events are mapped in CosNotification Structured Events as follows:

Structured Event Fields	Value
`domain_name`	Always set to `"TMEVT"`
`type_name`	Empty string
`event_name`	Empty string
Variable Header (Priority)	Empty sequence
Filterable Body Fields	Same as FML field name **Note:** Filterable body fields consist of name/value pair, where the name portion is the same as the FML field name.
Remainder of Body	Always set to void

The Oracle Tuxedo system detects and posts certain predefined events related to system warnings and failures. For example, system-generated events report on configuration changes, state changes, connection failures, and machine partitioning.

In order for an Oracle Tuxedo application to receive events posted by an Oracle Tuxedo application, it is necessary to understand how a FML buffer containing an Oracle Tuxedo event is used to fabricate an Oracle Tuxedo structured event. It is also necessary to know how the `domain_name` and `type_name` are related to the Oracle Tuxedo event name. There are two cases to consider: system events and user events.

Note that Oracle Tuxedo uses a leading dot (".") in the event name to distinguish system-generated events from application-defined events. An example of a system event is `.SysNetworkDropped`. An example of a user event is `eventsdropped`. To subscribe to these events, the Notification Service subscriber application must define the subscription as follows:

- System event

domain_name ="TMEVT"
type_name=".SysNetworkDropped"

- User event

  domain_name ="TMEVT"
  type_name="eventsdropped"

  When the events are received, the Notification Service subscriber application parses each event as follows:

  domain_name="TMEVT"
  type_name=""
  event_name=""
  variable_header=empty
  Filterable_data=(content of the FML buffer)

Parameters Used When Creating Subscriptions

When you create subscriptions, you can specify the following parameters. These parameters support the Oracle Simple Events API and the CosNotification Service API.

`subscription_name`
: Specifies a name that identifies the subscription to the Notification Service and the subscriber. Applications should use names that are meaningful to a system administrator since this is the primary way that an administrator associates an application with a subscription and the events that are delivered to the subscriber via the subscription. This parameter is optional (that is, an empty string can be passed in). More than one subscription can use the same name.

  The `subscription_name` must not exceed 128 characters in length.

`domain_type`
: Same parameter as the `domain_type` field in the Fixed Header portion of a structured event, as defined by the CORBA-based Notification Service specification. This field is a string that is used to identify a particular vertical industry domain in which the event type is defined, for example, "Telecommunications", "Finance", and "Health Care". Because this parameter is a regular expression, you can also use it to set domain patterns on which to filter. For example, to subscribe to all domains that begin with the letter F, set the domain to `"F.*"`. For information on how to construct regular expressions, see the `recomp` command in the *Oracle Tuxedo ATMI C Function Reference*.

`type_name`

Same parameter as the `type_name` field in the Fixed Header portion of a structure event, as defined in the CORBA-based Notification Service specification. It is a string that categorizes the type of event, uniquely within the domain, for example, Comm_alarm, StockQuote, and VitalSigns. Because this parameter is a regular expression, you can also use it to set event type patterns on which to filter. For example, to subscribe to all event types that begin with the letter F, you would set the type to `"F.*"`. For information on how to construct regular expressions, see the `recomp` command in the *Oracle Tuxedo ATMI C Function Reference*.

`data_filter`

Specifies the values of the fields of filterable data and variable headers on which you want to filter. For example, a subscription to news stories may have a domain of "News", a type of "Sports", and a data_filter of "Scores > 20".

This parameter defines the data that the subscription must match in Boolean expressions. The following data types are supported: `short`, `long`, `char`, `float`, `double`, and `string`. Table 2-2 lists the Boolean expression operators that are supported.

Table 2-2 Boolean Expression Operators

Expression	Operators
unary	+, -, !, ~
multiplicative	*, /, %
additive	+, -
relational	< , >, <=, >=, ==, !=
equality and matching	==, !=, %%, !%
exclusive OR	^
logical AND	&&
logical OR	||

To use data filtering, you must set up an FML table, include filters in the subscription, filter the data, and post the event. Listing 2-3 shows an example of these tasks.

Listing 2-3 Data Filtering Requirements

```
//Setting up the FML Table

Field table file.
----------------
*base 2000

*Field Name    Field #   Field Type   Flags    Comments
-----------    -------   ---------    ------   ------
StockName        1        string        -        -
PricePerShare    2        double        -        -
CustomerId       3        long          -        -
CustomerName     4        string        -        -

//Subscription data filtering.
1) "NumberOfShares > 100 && NumberOfShares < 1000"
2) "CustomerId == 3241234"
3) "PricePerShare > 125.00"
4) "StockName == 'BEAS'"
5) "CustomerName %% '.*Jones.*'" // CustomerName contains "Jones"
6) "StockName == 'BEAS' && PricePerShare > 150.00"

//Posting the event.
// C++
CosNotification::StructuredEvent ev;
...
ev.filterable_data[0].name    = CORBA::string_dup("StockName");
ev.filterable_data[0].value <<= "BEAS";
ev.filterable_data[1].name    = CORBA::string_dup("PricePerShare");
ev.filterable_data[1].value <<= CORBA::Double(175.00);
ev.filterable_data[2].name    = CORBA::string_dup("CustomerId");
ev.filterable_data[2].value <<= CORBA::Long(1234567);
ev.filterable_data[3].name    = CORBA::string_dup("CustomerName");
ev.filterable_data[3].value <<= "Jane Jones";
```

For more information about filter grammar, see "Creating FML Field Table Files for Events" on page -7 and the section "Boolean Expression of fielded Buffers" in *Programming Oracle Tuxedo ATMI Applications Using FML*.

`push_consumer`

Identifies the callback object that will be used by the Notification Service to deliver a structured event. Subscriber applications must implement the CosNotifyComm::StructuredPushConsumer interface so that the Notification Service can call it to deliver events.

Note: You can use either transient or persistent object references for the callback objects. Both QoS and application run times should be taken into consideration when deciding which type of object reference to use. For information to assist you in deciding which type of object reference to use, refer to Table 2-3.

Table 2-3 When to Use Transient Versus Persistent Object References for Joint Client/Servers

If the subscription ...	Then ...
Will have a transient QoS and will start and shut down once.	You should use a transient object reference. It this case, Oracle Systems, Inc. recommends the subscriber application unsubscribe on shutdown so as to release system resources, however, this is not a requirement.
Will have a persistent QoS and will start and shut down once.	You should use a transient object reference.
Will have a persistent Qos and will start and shut down multiple times.	You must use a persistent object reference and store the host and port so the same host and port is used each time the subscriber shuts down and restarts. In this case, use of the bidirectional IIOP feature is not recommended. **Note:** If a joint client/server is used, it must be remote (outside the Oracle Tuxedo domain) because persistent object references are not supported inside the domain.
Will have a transient QoS and will start and shut down multiple times.	You can use a persistent object reference; however, Oracle Systems, Inc. does not recommend this configuration unless you can guarantee that no events for this subscriber will be posted while the subscriber is shut down.

`qos` **`(quality of service)`**

Specifies the desired quality of service of the subscription. It can take one of two values: transient or persistent.

For transient subscriptions, the Notification Service makes only one attempt to deliver the event to a subscriber. If that attempt fails, the event is discarded and, if the Notification Service does not receive the `CORBA::TRANSIENT` exception, it concludes that the subscriber is shutdown or otherwise not available and cancels the subscription. If the Notification Service receives the `CORBA::TRANSIENT` exception when an attempt to deliver fails, it assumes that the subscriber is busy and discards the event, but it does not cancel the subscription.

For persistent subscriptions, if the first delivery attempt fails, the Notification Service holds the event in the pending queue and keeps attempting to deliver the subscription until the configurable retry limit is reached. When the retry limit is reached, the Notification Service moves the event on an error queue where it is held for disposition by the system administrator. The system administrator either removes the event from the error queue, which in effect discards it, or moves it back to the pending queue so that further attempts to deliver it can be made.

Note: For persistent subscriptions, the Notification Service always does a two-way invoke on callback objects to deliver events. If a joint client/server does not activate a callback object (the event receiver) before it calls `orb->run` and then the Notification Service invokes on the callback object, as far as the POA is concerned, the callback object does not exist. In this case `CORBA::OBJECT_NOT_EXIST` exception is returned. If the Notification Service receives a `CORBA::OBJECT_NOT_EXIST` exception, it drops the subscription and the event; otherwise, the subscription is retained and the event is retried.

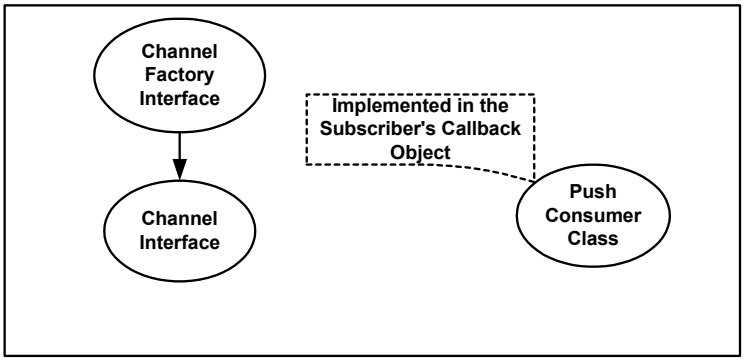
Oracle Simple Events API

Simplicity and ease-of-use are the defining characteristics of the Oracle Simple Events application programming interface (API). Its capabilities are similar to those of the Oracle Tuxedo EventBroker.

The Oracle Simple Events API consists of the following interfaces (see Figure 2-2):

- `Tobj_SimpleEvents::Channel`
- `Tobj_SimpleEvents::ChannelFactory`
- `CosNotifyComm::StructuredPushConsumer`

Figure 2-2 Oracle Simple Events Interfaces



The `Tobj_SimpleEvents::Channel` and the `Tobj_SimpleEvents::ChannelFactory` interfaces are implemented by the Notification Service and are described below.

The `CosNotifyComm::StructuredPushConsumer` interface is implemented by the subscribers. For a description of this interface, see "CosNotifyComm::StructuredPushConsumer::push_structured_event" on page -51.

Note: The CosNotification Service classes referred to in this section are fully described in the CosNotification Service IDL files, which are located in the `tuxdir/include` directory.

Note: If you use class operations that are not supported, the `CORBA::NO_IMPLEMENT` exception is raised.

TOBJ_SimpleEvents::Channel Interface

The Channel interface is used:

- By subscribers to subscribe and unsubscribe to events and to determine if a subscription exists
- By posters to post events to the Notification Service

This interface provides these operations:

- `subscribe()`
- `unsubscribe()`
- `exists()`

- push_structured_event()

The CORBA IDL for this interface:

```
module Tobj_SimpleEvents
  {
  typedef long   SubscriptionID;
  typedef string RegularExpression;
  typedef string FilterExpression;

  const SubscriptionType TRANSIENT_SUBSCRIPTION = 0;
  const SubscriptionType PERSISTENT_SUBSCRIPTION = 1;

interface Channel
    {
     void push_structured_event(
         in  CosNotification::StructuredEvent   event);

     SubscriptionID subscribe (
       in    string                            subscription_name,
       in    RegularExpression                 domain,
       in    RegularExpression                 type,
       in    FilterExpression                  data_filter,
       in    CosNotification::QoSProperties    qos,
       in    CosNotifyComm::StructuredPushConsumer push_consumer);

     boolean exists( in SubscriptionID id );

     void unsubscribe( in SubscriptionID  id );
    };
  };
```

These operations are described in the following section.

Channel::subscribe

CORBA IDL

```
SubscriptionID subscribe (
      in    string                            subscription_name,
      in    RegularExpression                 domain,
      in    RegularExpression                 type,
      in    FilterExpression                  data_filter,
```

```
        // The filter expression must length 1 and the name must
        // be TRANSIENT_SUBSCRIPTION or PERSISTENT_SUBSCRIPTION.
        in    CosNotification::QoSProperties     qos,
        in    CosNotifyComm::StructuredPushConsumer push_consumer
);
```

Parameters

For a description of the parameters supported by this operation, see “Parameters Used When Creating Subscriptions” on page -11.

Exceptions

`CORBA::BAD_PARAM`

Indicates one of the following problems:

`Tobj_Events::SUB_INVALID_FILTER_EXPRESSION`
`Tobj_Events::SUB_UNSUPPORTED_QOS_VALUE`

`CORBA::IMP_LIMIT`

Indicates one of the following problems:

`Tobj_Events::SUB_DOMAIN_BEGINS_WITH_SYSEV`
`Tobj_Events::SUB_EMPTY_DOMAIN`
`Tobj_Events::SUB_EMPTY_TYPE`
`Tobj_Events::SUB_DOMAIN_AND_TYPE_TOO_LONG`
`Tobj_Events::SUB_FILTER_TOO_LONG`
`Tobj_Events::SUB_NAME_TO_LONG`
`Tobj_Events::TRANSIENT_ONLY_CONFIGURATION`

`CORBA::INV_OBJREF`

Indicates the following problem:

`Tobj_Events::SUB_NIL_CALLBACK_REF`

Note: For more information on exceptions and corresponding minor codes, see “Exception Minor Codes” on page -53.

Description

Use this operation to subscribe to events. This operation is called by a subscriber application on the Notification Service to create a subscription to a particular event. The subscription name, domain name, type name, data filter, quality of service, and the object reference of the subscriber’s callback object are passed in. The callback object implements the CosNotifyComm::StructuredPushConsumer IDL interface.

Note: For subscribers that shut down and restart, you must write the `subscription_id` to persistent storage.

To use data filtering or subscribe to Oracle Tuxedo system events or events posted by an Oracle Tuxedo application, see the sections “Creating FML Field Table Files for Events” on page -7 and “Interoperability with Oracle Tuxedo Applications” on page -9.

Return Value

Returns a unique subscription identifier. The effect of this operation is not instantaneous. There can be a delay between returning from this operation and the actual start of event delivery. The length of the delay period may be significant depending on your configuration. For more information on factors impacting this delay period, see “Synchronizing Databases” on page -23.

Note: Notification Service applications that start and shut down only once can use the `subscription_id` to determine if their subscription has been cancelled automatically or by the system administrator.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see “Creating a Subscription” on page -8.

C++ code example:

```
subscription_id = channel->subscribe(
          subscription_name,
          "News", // domain
          “Sports”,   // type
          "",     // No data filter.
          qos,
          news_consumer.in()
        );
```

Channel::unsubscribe

CORBA IDL

```
void unsubscribe( in SubscriptionID id );
```

Parameter

`subscription_id`
: The subscription identifier.

Exceptions

`CORBA::BAD_PARAM`
: Indicates the following problem: `Tobj_Events::INVALID_SUBSCRIPTION_ID`

Note: For more information on exceptions and corresponding minor codes, see "Exception Minor Codes" on page -53.

Description

Used to unsubscribe. Subscriber applications use this operation to terminate subscriptions. On return from this operation, no further events can be delivered. There is one input parameter: `SubscriptionID`, which you got when you subscribed.

Note: This operation is not instantaneous. After returning from this operation, a subscriber may continue to receive events for a period of time. The period of time may be significant depending on your configuration. For more information on factors impacting this period of time, see "Synchronizing Databases" on page -23.

Examples

C++ code example:

```
channel->unsubscribe(subscription_id);
```

Channel::push_structured_event

CORBA IDL

```
void push_structured_event(
      in   CosNotification::StructuredEvent   notification
);
```

Parameter

`notification`
: This parameter contains the structured event as defined by the CosNotification Service specification.

Exceptions

`CORBA_IMP_LIMIT`
: Indicates one of the following problems with the subscription:
`Tobj_Events::POST_UNSUPPORTED_VALUE_IN_ANY`
`Tobj_Events::POST_UNSUPPORTED_PRIORITY_VALUE`
`Tobj_Events::POST_DOMAIN_CONTAINS_SEPARATOR`

```
Tobj_Events::POST_TYPE_CONTAINS_SEPARATOR
Tobj_Events::POST_SYSTEM_EVENTS_UNSUPPORTED
Tobj_Events::POST_EMPTY_DOMAIN
Tobj_Events::POST_EMPTY_TYPE
Tobj_Events::POST_DOMAIN_AND_TYPE_TOO_LONG
```

Note: For more information on exceptions and corresponding minor codes, see "Exception Minor Codes" on page -53.

Description

Used by the poster application to post an event to the Notification Service.

Note: This operation has transactional behavior when used in the context of a transaction. For more information, see the section "Using Transactions" on page -4.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating and Posting Events" on page -3.

C++ code example:

```
channel->push_structured_event(notification);
```

Channel::exists

CORBA IDL

```
boolean exists(in SubscriptionID subscription_id);
```

Parameter

`subscription_id`
The subscription identifier.

Exceptions

`CORBA::BAD_PARAM`
Indicates the following problem: `Tobj_Events::INVALID_SUBSCRIPTION_ID`

If the `subscription_id` is for a subscription created using the CosNotification Service API, this exception is always returned.

Note: For more information on exceptions and corresponding minor codes, see "Exception Minor Codes" on page -53.

Description

Used by subscriber applications to determine if a subscription exists. Since the system administrator can delete subscriptions manually and the Notification Service can delete transient subscriptions automatically, a subscriber application might want to use this operation so that it can recreate the subscription, if necessary. The `subscription_id` used in this operation is the same one that you got when you subscribed.

Return Value

Returns Boolean True of the subscription exists and False if it does not.

Examples

C++ code example:

```
if channel->exists (subscription_id) {
    // The subscription is still valid.
} else {
  // The subscription no longer exists.
}
```

TOBJ_SimpleEvents::ChannelFactory Interface

The `ChannelFactory` interface is used to find event channels. This interface provides a single operation: `find_channel`.

The CORBA IDL for this interface:

```
module Tobj_SimpleEvents
{
  typedef long   ChannelID;

interface ChannelFactory
{
 Channel find_channel(
   in ChannelID channel_id // Must be DEFAULT_CHANNEL
  );
 };
};
```

Channel_Factory::find_channel

CORBA IDL

```
Channel find_channel(
      in  ChannelID  channel_id );
```

Parameter

In this release of Oracle Tuxedo, there can only be one event channel; therefore, the `ChannelID` that is passed in must be set to `Tobj_SimpleEvents::DEFAULT_CHANNEL` (for C++).

Exceptions

`CORBA::BAD_PARAM`
Indicates the following problem:
`Tobj_Events::INVALID_CHANNEL_ID`

Note: For more information on exceptions and corresponding minor codes, see "Exception Minor Codes" on page -53.

Description

Used by poster applications and subscriber applications. This operation is used to find the event channel so that it can be used by the poster to post events and by the subscriber to subscribe and unsubscribe to events.

Return Value

Returns the default event channel's object reference.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Getting the Event Channel" on page -2.

C++ code example:

```
channel_factory->find_channel(
               Tobj_SimpleEvents::DEFAULT_CHANNEL);
```

CosNotification Service API

This section contains a discussion of the operations defined by the CosNotification Service that are implemented by the Oracle Tuxedo CORBA Notification Service. These operations are only

a subset of the complete set of operations. This subset is a functionally complete API that can be used as an alternative to the Oracle Simple Events API.

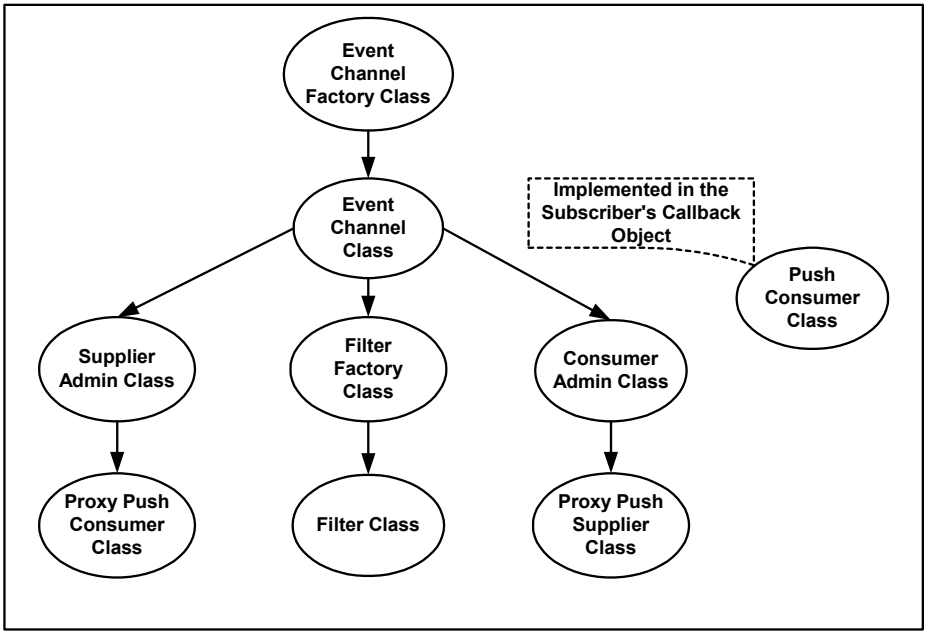
This API is more complex then the Oracle Simple Events API. There are two reasons for this. First, the CosNotification Service API is more complex. Second, the Oracle Tuxedo implementation of the CosNotification Service API places additional restrictions on the operations that are supported. Because this complexity offers no advantages in terms of performance or flexibility, Oracle Systems, Inc. recommends that you use the Oracle Simple Events API whenever possible.

The CosNotification API is provided for those who require that a standard API be used whenever possible for purposes of portability. In regard to functionality, this API provides no benefits beyond those offered by the Simple Events API. Applications that are developed using this API will be mostly, but not completely, portable. The reason for this is that not enough of the CosNotification Service API is supported to facilitate portability. For example, the filtering grammar required by the CORBA-based Notification Service is based on the COS Trader grammar. Since Oracle Tuxedo does not support this grammar, but supports an alternative grammar based on the Oracle Tuxedo EventBroker grammar, any application that requires filtering will not be portable. The same is true for QoS, that is, the CosNotification Service API does not support the CORBA-based Notification Service standard qualities of service, but it does support alternative qualities of service.

Overview of Supported CosNotification Service Classes

Figure 2-3 shows the CosNotification Service classes implemented, in full or in part, in this release of Oracle Tuxedo and their relationships.

Figure 2-3 Implemented CosNotification Service Classes



The operations supported by each class are summarized below. For more detailed descriptions, see “Detailed Descriptions of CosNotification Service Classes” on page -27.

- CosNotifyChannelAdmin::EventChannelFactory Class

  This class is used by the event poster and subscriber applications. It supports the `get_channel_factory` operation which is used to get the channel factory when posting, subscribing, and unsubscribing to events.

- CosNotifyChannelAdmin::EventChannel Class

  This class is used by event poster and subscriber applications. It supports three operations:

  - `default_consumer_admin`—used by event subscriber applications to get the consumer admin object.
  - `default_supplier_admin`—used by event poster applications to get the supplier admin object.

- `default_filter_factory`—used by event subscriber applications to get the filter factory object.

- CosNotifyChannelAdmin::SupplierAdmin Class

  This class is used by event poster applications. It supports the `obtain_notification_push_consumer` operation. Poster applications use this operation to create proxy push consumer objects which in turn are used to post events to the Notification Service.

- CosNotifyChannelAdmin::StructuredProxyPushConsumer Class

  This class is used by event poster applications. It supports the following operations:

  - `connect_structured_push_supplier`—used by event poster applications to connect the proxy push supplier to the Notification Service event channel.
  - `push_structured_event`—used by event poster applications to post the event to the Notification Service event channel.
  - `disconnect_structured_push_consumer`—used by event poster applications to disconnect the proxy push supplier from the Notification Service event channel.

- CosNotifyFilter::FilterFactory Class

  This class is used by event subscriber applications to create a filter object. It supports the `create_filter` operation. The filter object provides all data filtering including domain, type, and filterable data.

- CosNotifyFilter::Filter Class

  This class is used by event subscriber applications. It supports the following operations:

  - `add_contraints` operation—used to set the filter's domain, type, and data filter.
  - `destroy` operation—used to destroy the filter object.

- CosNotifyChannelAdmin::ConsumerAdmin Class

  This class is used by event subscriber applications. It supports the following operations:

  - `obtain_notification_push_supplier`—used by event subscriber applications to create proxy push supplier objects which in turn are used to deliver events to the subscriber's callback object.
  - `get_proxy_supplier`—used by event subscriber applications to retrieve the object reference for the proxy push supplier object. This operation is only used when the subscriber application shuts down then restarts and cancels the subscription. This is

because subscribers need to discard the object reference from the first run and get it back again for the next run. Subscribers cannot reuse object references from one run to the next.

- CosNotifyChannelAdmin::StructuredProxyPushSupplier Class

  This class is used by event subscriber applications. It supports the following operations:

  - `connect_structured_push_consumer`—used by event subscriber applications to connect the subscriber to the proxy push supplier.
  - `set_qos`—used by event subscriber applications to set the quality of service for subscriptions.
  - `add_filter`—used by event subscriber applications to add the filter object to the subscription.
  - `get_filter`—used by event subscriber applications when performing unsubscribe operations to get the filter associated with the subscription. This operation is only used when the subscriber application shuts down then restarts.
  - `disconnect_structured_push_supplier`—used by event subscriber applications to unsubscribe.

- CosNotifyComm::StructuredPushConsumer

  This interface is implemented by event subscriber applications. It supports the `push_structured_event` operation. The Notification Service invokes this operation to deliver events to the subscriber.

Detailed Descriptions of CosNotification Service Classes

This section describes the CosNotification Service classes that this release of Oracle Tuxedo implements. These classes are fully described in the CosNotification Service IDL files, which are located in the `tuxdir/include` directory.

Note: If you use class operations that are not supported, the `CORBA::NO_IMPLEMENT` exception is raised.

CosNotifyFilter::Filter Class

This class is used by event subscriber applications. The OMG IDL for this class is as follows:

```
Module CosNotifyFilter
{
interface Filter {
```

```
    ConstraintInfoSeq add_constraints (
            in ConstraintExpSeq constraint)
        raises (InvalidConstraint);

    void destroy();
  };
}; //CosNotifyFilter
```

CosNotifyFilter::Filter::add_constraints

Synopsis

Sets the domain, type, and data filter parameters on the filter object.

OMG IDL

```
ConstraintInfoSeq add_constraints (
            in ConstraintExpSeq constraint)
        raises (InvalidConstraint);
```

Exceptions

`CosNotifyFilter::InvalidConstraint`
Never raised.

`CORBA::BAD_PARAM`
Indicates the following problem: `Tobj_Events::SUB_INVALID_FILTER_EXPRESSION.`

`CORBA_IMP_LIMIT`
Indicates one of the following problems:

```
Tobj_Notification::SUB_ADD_CONS_ON_TIMED_OUT_FILTER
Tobj_Notification::SUB_MULTIPLE_CALLS_TO_ADD_CONS
Tobj_Notification::SUB_MULTIPLE_CONSTRAINTS_IN_LIST
Tobj_Notification::SUB_MULTIPLE_TYPES_IN_CONSTRAINT
Tobj_Notification::SUB_SYSTEM_EVENTS_UNSUPPORTED
Tobj_Events::SUB_DOMAIN_BEGINS_WITH_SYSEV
Tobj_Events::SUB_EMPTY_DOMAIN
Tobj_Events::SUB_EMPTY_TYPE
Tobj_Events::SUB_FILTER_TOO_LONG
```

Note: For more information on exceptions and corresponding minor codes, see “Exception Minor Codes” on page -53.

Description

Used when subscribing. This operation is used in subscriber applications to define the kind of event to which you want to subscribe. You set the domain, type, and data filter parameters on the filter object. For a description of these parameters, see “Parameters Used When Creating Subscriptions” on page -11.

Note: The Oracle Tuxedo implementation of the `add_constraints` operation (1) can only be called once, (2) must be called before the filter is added to the proxy object, and (3) must consist of only a single constraint that has a single event type.

Return Value

Returns `an empty list`, which we recommend that the caller ignores.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see “Creating a Subscription” on page -10.

C++ code example:

```
// set the filtering parameters
// (domain = "News", type, and no data filter)
CosNotifyFilter::ConstraintExpSeq constraints;
constraints.length(1);
constraints[0].event_types.length(1);
constraints[0].event_types[0].domain_name =
    CORBA::string_dup("News");
constraints[0].event_types[0].type_name =
                CORBA::string_dup (“Sports”);
// no data filter
constraints[0].constraint_expr = CORBA::string_dup("");
CosNotifyFilter::ConstraintInfoSeq_var
add_constraints_results = // ignore this returned value
     filter->add_constraints(constraints);
```

CosNotifyFilter::Filter::destroy

Synopsis

Destroys the filter object.

OMG IDL

```
void destroy();
```

Exceptions

`CORBA::BAD_PARAM`

Indicates the following problem: `Tobj_Events::SUB_INVALID_FILTER_EXPRESSION`.

Note: For more information on exceptions and corresponding minor codes, see "Exception Minor Codes" on page -53.

Description

Used when unsubscribing. This operation is used in subscriber applications to destroy the target filter object.

Note: Do not destroy the filter object until you are ready to cancel the corresponding subscription.

CosNotifyFilter::FilterFactory Class

This class is used by event subscriber applications. The OMG IDL for this class is as follows:

```
Module CosNotifyFilter
{
interface FilterFactory {
   Filter create_filter (
          in string constraint_grammar)
      raises (InvalidGrammar);
   destroy();
  };
}; //CosNotifyFilter
```

CosNotifyFilter::FilterFactory::create_filter

Synopsis

Determines which events are delivered to a subscription.

OMG IDL

```
Filter create_filter (
          in string constraint_grammar)
      raises (InvalidGrammar);
```

Exceptions

`CosNotifyFilter::InvalidGrammar`
Indicates the `constraint_grammar` is not supported.

Description

Used in the subscriber application to create a new filter object. This filter is used to determine which events are delivered to a subscription. The subscriber must set up the filter and add it to the proxy within five minutes; otherwise, the filter will be destroyed. The filter grammar must be set to `Tobj_Notification::Constraint_grammar`; otherwise, the `InvalidGrammar` exception is raised.

Return Value

Returns the new filter's object reference.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating a Subscription" on page -10.

C++ code example:

```
filter_factory->create_filter(
      Tobj_Notification::CONSTRAINT_GRAMMAR
);
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier Class

This class is used by event subscriber applications. The OMG IDL for this class is as follows:

```
Module CosNotifyChannelAdmin
{
 interface StructuredProxyPushSupplier :
       ProxySupplier,
       CosNotifyComm::StructuredPushSupplier {

    void connect_structured_push_consumer (
           in CosNotifyComm::StructuredPushConsumer push_consumer)
         raises(CosEventChannelAdmin::AlreadyConnected,
                CosEventChannelAdmin::TypeError );
    };
    // The following operations are inherited.
    void set_qos(in QoSProperties qos)
```

```
        raises (UnsupportedQoS);
    FilterID add_filter (in Filter new_filter );
    Filter get_filter( in FilterID filter )
        raises ( FilterNotFound);
    void disconnect_structured_push_supplier();
    readonly attribute ProxyType     MyType;
  };
}; //CosNotifyChannelAdmin
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier:: connect_structured_push_consumer

Synopsis

Completes a subscription.

OMG IDL

```
void connect_structured_push_consumer (
       in CosNotifyComm::StructuredPushConsumer push_consumer)
     raises(CosEventChannelAdmin::AlreadyConnected,
            CosEventChannelAdmin::TypeError );
```

Exceptions

`CosEventChannelAdmin::TypeError`
: Never raised.

`CORBA::INV_OREF`
: `Tobj_Events::SUB_NIL_CALLBACK_REF`

`CORBA::IMP_LIMIT`
: Indicates one of the following problems:
`Tobj_Events::SUB_DOMAIN_AND_TYPE_TOO_LONG`
`Tobj_Events::SUB_NAME_TO_LONG`
`Tobj_Events::TRANSIENT_ONLY_CONFIGURATION`
`Tobj_Notification::SUBSCRIPTION_DOESNT_EXIST.`

`CORBA::OBJECT_NOT_EXIST`
: The proxy does not exist.

`CosEventChannelAdmin::AlreadyConnected`
: Indicates that the **`connect_structured_push_consumer`** operation has already been invoked.

Note: For exception definitions and corresponding minor codes, see "Exception Minor Codes" on page -53.

Description

Use this operation when subscribing. This operation is used in subscriber applications to subscribe to events. The `push_consumer` parameter identifies the subscriber's callback object.

Once the **`connect_structured_push_consumer`** has been called, the Notification Service will proceed to send events to the subscriber by invoking the callback object's `push_structured_event` operation. If the **`connect_structured_push_consumer`** has already been called, the `AlreadyConnected` exception is raised.

Note: You must call `set_qos` and `add_filter` before calling **`connect_structured_push_consumer`**.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating a Subscription" on page -10.

C++ code example:

```
subscription->connect_structured_push_consumer(
    news_consumer.in()
);
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier::set_qos

Synopsis

Sets the QoS for the subscription.

OMG IDL

```
void set_qos(in QoSProperties qos)
         raises (UnsupportedQoS);
```

Exceptions

`UnsupportedQoS`
: Never raised.

`ORBA::IMP_LIMIT`

Indicates one of the following problems:

`Tobj_Notification::SUB_MULTIPLE_CALLS_TO_SET_QOS`
`Tobj_Notification::SUB_CANT_SET_QOS_AFTER_CONNECT`
`Tobj_Notification::SUBSCRIPTION_DOESNT_EXIST`
`Tobj_Notification::SUB_UNSUPPORTED_QOS_VALUE`

Note: For more information on exceptions and corresponding minor codes, see "Exception Minor Codes" on page -53.

Description

Used when subscribing. This operation is used in subscriber applications to set the QoS for the subscription. It takes as an input parameter a sequence of name-value pairs which encapsulates quality-of-service property settings that the subscriber is requesting.

There are two components of the QoS: the subscription type and the subscription name. The subscription type is set by constructing a name-value pair where the name is `Tobj_Notification::SUBSCRIPTION_TYPE` and the value is either `Tobj_Notification::PERSISTENT_SUBSCRIPTION`, or `Tobj_Notification::TRANSIENT_SUBSCRIPTION`. For more information and additional usage details, see "Quality of Service" on page -2.

The subscription name is set by constructing a name-value pair, where the name is `Tobj_Notification::SUBSCRIPTION_NAME`, and the value is a user-defined string.

For more information on this parameter, see "Parameters Used When Creating Subscriptions" on page -11.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating a Subscription" on page -10.

C++ code example:

```
CosNotification::QoSProperties qos;
qos.length(2);
qos[0].name =
    CORBA::string_dup(Tobj_Notification::SUBSCRIPTION_NAME);
qos[0].value <<= "MySubsription";
qos[1].name =
    CORBA::string_dup(Tobj_Notification::SUBSCRIPTION_TYPE);
qos[1].value <<=
    Tobj_Notification::TRANSIENT_SUBSCRIPTION;
```

```
subscription->set_qos(qos);
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier::add_filter

Synopsis

Sets the filter object on the subscriber's callback object.

OMG IDL

```
add_filter(
      in Filter new_filter
);
```

Exceptions

`CORBA::IMP_LIMIT`

Indicates one of the following problems:

`Tobj_Notification::SUB_MULTIPLE_CALLS_TO_SET_FILTER`
`Tobj_Notification::SUB_ADD_FILTER_AFTER_CONNECT`
`Tobj_Notification::SUB_NIL_FILTER_REF`
`Tobj_Notification::SUB_NO_CUSTOM_FILTERS`

`CORBA::OBJECT_NOT_EXIST`

Indicates that the subscription does not exist.

Note: For more information on exceptions and corresponding minor codes, see "Exception Minor Codes" on page -53.

Description

Used when subscribing. This operation is used in subscriber applications to set the filter object to the subscriber's callback object. If the application using this operation will be shut down and restarted, the `filter_id` should be written to persistent storage.

Note: This operation: (1) cannot be called after the subscriber callback object is connected (see `connect_structured_push_consumer` above), (2) cannot be called more than once, and (3) when it is called, the filter constraint expression must already be present in the filter (see `CosNotifyFilter::Filter add_constraints`).

Note: Only filters created by the event channel's default filter factory can be added.

Return Value

`Returns a filter_id`.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating a Subscription" on page -10.

C++ code example:

```
CosNotifyFilter::FilterID filter_id =
      subscription->add_filter(filter.in());
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier::get_filter

Synopsis

Gets an object reference to the filter currently associated with the subscriber's callback object.

OMG IDL

```
Filter get_filter( in FilterID filter )
      raises ( FilterNotFound);
```

Exceptions

`CosNotifyChannelAdmin::FilterNotFound`
The filter could not be found.

Description

Used when a restartable subscriber wants to unsubscribe. This operation is used in subscriber applications to get an object reference to the filter currently associated with the subscriber's callback object. The `FilterID` that is passed in must be valid for the subscriber's StructuredProxyPushSupplier object. If the `FilterID` is not valid for any proxy object associated with the event channel, then a `FilterNotFound` exception is thrown. The operation is only used by subscribers that shut down and restart.

Restrictions

The following usage restrictions and guidelines apply to this operation:

a. Filter object references that are returned from this operation cannot be used in comparison operations.

b. Filter object references returned by this operation can be used by the `CosNotifyFilter::Filter::destroy` operations but are of little use since they cannot be modified or added to proxy objects.

Return Value

Returns a filter object reference to the filter currently associated with the subscriber's callback object.

Examples

C++ code example:

```
CosNotify::Filter_var filter =
     subscription->get_filter( filter_id() );
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier:: disconnect_structured_push_supplier

Synopsis

`Used to unsubscribe.`

OMG IDL

```
void disconnect_structured_push_supplier();
```

Exceptions

`CORBA::OBJECT_NOT_EXIST`

Indicates that the subscription to be disconnected does not exist.

Note: For more information on exceptions and corresponding minor codes, see "Exception Minor Codes" on page -53.

Description

Used by subscriber applications when unsubscribing. This operation is used in subscriber applications to terminate a connection between the Notification Service and the subscriber's callback object.

Note: This operation does not stop event delivery instantaneously. After returning from this operation, a subscriber may continue to receive events for a period of time.

Examples

C++ code example:

```
subscription->disconnect_structured_push_supplier();
```

CosNotifyChannelAdmin::StructuredProxyPushSupplier::MyType

Synopsis

Always returns `CosNotifyChannelAdmin::PUSH_STRUCTURED` proxy.

OMG IDL

```
readonly attribute ProxyType MyType
```

Description

Always returns `CosNotifyChannelAdmin::PUSH_STRUCTURED` proxy.

CosNotifyChannelAdmin::StructuredProxyPushConsumer Class

This class is used by event posting applications. The OMG IDL for this class is as follows:

```
Module CosNotifyChannelAdmin
{
 interface StructuredProxyPushConsumer :
       ProxyConsumer,
       CosNotifyComm::StructuredPushConsumer {

    void connect_structured_push_supplier (
          in CosNotifyComm::StructuredPushSupplier push_supplier)
       raises(CosEventChannelAdmin::AlreadyConnected);
    // The following operations are inherited.
    readonly attribute MyType;
    void push_structured_event(
          in CosNotification::StructuredEvent notification )
       raises( CosEventComm::Disconnected );
    void disconnect_structured_push_consumer();
   };
}; \\StructuredProxyPushConsumer
```

CosNotifyChannelAdmin::StructuredProxyPushConsumer:: connect_structured_push_supplier

Synopsis

```
Prepares the Notification Service to receive an event.
```

OMG IDL

```
void connect_structured_push_supplier (
          in CosNotifyComm::StructuredPushSupplier push_supplier)
       raises(CosEventChannelAdmin::AlreadyConnected);
```

Exception

`CosEventChannelAdmin::AlreadyConnected`
Never raised.

Description

Used by poster applications when posting events. You must call this operation to prepare the Notification Service to receive an event and you must pass in a NIL when you use this operation. The sequence of usage is as follows:

1. Make a proxy.
2. Use this operation to connect to the Notification Service and pass in a NIL.
3. Post events.
4. Before exiting the poster program, disconnect.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating and Posting Events" on page -3.

C++ code example:

```
proxy_push_consumer->connect_structured_push_supplier(
      CosNotifyComm::StructuredPushSupplier::_nil()
    );
```

CosNotifyChannelAdmin::StructuredProxyPushConsumer:: push_structured_event

Synopsis

Posts events to the event channel.

OMG IDL

```
void push_structured_event(
        in CosNotification::StructuredEvent notification )
    raises( CosEventComm::Disconnected );
```

Exceptions

`CosEventComm::Disconnected`

Never raised.

`CORBA::IMP_LIMIT`

Indicates one of the following problems:

`Tobj_Events::POST_UNSUPPORTED_VALUE_IN_ANY`
`Tobj_Events::POST_UNSUPPORTED_PRIORITY_VALUE`
`Tobj_Events::POST_DOMAIN_CONTAINS_SEPARATOR`
`Tobj_Events::POST_TYPE_CONTAINS_SEPARATOR`
`Tobj_Events::POST_SYSTEM_EVENTS_UNSUPPORTED`
`Tobj_Events::POST_EMPTY_DOMAIN`
`Tobj_Events::POST_EMPTY_TYPE`
`Tobj_Events::POST_DOMAIN_AND_TYPE_TOO_LONG`

Note: For more information on exceptions and corresponding minor codes, see "Exception Minor Codes" on page -53.

Descriptions

Used when posting events. This operation is used in poster applications to post events to the event channel.

Note: This operation differs from the standard CORBA definition in the following ways:

a. The Priority in the variable header section of the event, if specified, must be `short` value in the range of 1 to 100.

b. If event filterable data filtering (versus filtering on domain and type only) is required, or if events are to be received by an Oracle Tuxedo subscriber, then additional restrictions apply. See "Structured Event Fields, Types, and Filters" on page -5 and "Interoperability with Oracle Tuxedo Applications" on page -9.

Note: This operation has transactional behavior when used in the context of a transaction. For more information, see "Using Transactions" on page 2-4.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating and Posting Events" on page -3.

C++ code example:

```
proxy_push_consumer->push_structured_event(notification);
```

CosNotifyChannelAdmin::StructuredProxyPushConsumer:: disconnect_structured_push_consumer

Synopsis

Stops posting events.

OMG IDL

```
void disconnect_structured_push_consumer();
```

Descriptions

Used when posting events. This operation is used by poster applications to stop posting events. It takes no input parameters and returns no values. The recommended usage sequence is as follows:

1. Make a proxy.
2. Connect and disconnect on every run of the poster application.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating and Posting Events" on page -3.

C++ code example:

```
proxy_push_consumer->disconnect_structured_push_consumer();
```

CosNotifyChannelAdmin::StructuredProxyPushConsumer::MyType

Synopsis

Always returns `CosNotifyChannelAmdmin::PUSH_STRUCTURED` proxy.

OMG IDL

```
readonly attribute ProxyType MyType
```

Description

Always returns `CosNotifyChannelAmdmin::PUSH_STRUCTURED` proxy.

CosNotifyChannelAdmin::ConsumerAdmin Class

This class is used by event subscriber applications. The OMG IDL for this class is as follows:

```
Module CosNotifyChannelAdmin
{
 interface ConsumerAdmin :

       CosNotification::QoSAdmin,
       CosNotifyComm::NotifySubscribe,
       CosNotifyFilter::FilterAdmin,
       CosEventChannelAdmin::ConsumerAdmin {

    ProxySupplier obtain_notification_push_supplier (
           in ClientType ctype,
           out ProxyID proxy_id)
         raises ( AdminLimitExceeded )

    ProxySupplier get_proxy_supplier (
           in ProxyID proxy_id )
        raises ( ProxyNotFound );

  };
}; //CosNotifyChannelAdmin
```

CosNotifyChannelAdmin::ConsumerAdmin:: obtain_notification_push_supplier

Synopsis

Creates proxy push supplier objects.

OMG IDL

```
ProxySupplier obtain_notification_push_supplier (
           in ClientType ctype,
```

```
        out ProxyID proxy_id)
      raises ( AdminLimitExceeded )
```

Exceptions

`CosNotifyChannelAdmin::AdminLimitExceeded`
Never raised.

`CORBA::IMP_LIMIT`
Indicates the following problem:
`Tobj_Notification::SUB_UNSUPPORTED_CLIENT_TYPE`

Description

Used when subscribing. This operation is used in subscriber applications to create proxy push supplier objects. Only structured events are supported (that is, `ANY_EVENT` and `SEQUENCE_EVENT ClientTypes` are not supported). Therefore, the `ClientType` input parameter must be set to `CosNotifyComm::STRUCTURED_EVENT`. If you shut down and restart the subscriber and subscription survives more than one run of your program, the `ProxyID` returned by this operation should be durably stored. The subscriber must narrow the proxy supplier to `CosNotifyChannelAdmin::StructuredProxyPushSupplier`. All required operations must be completed in five minutes.

Note: Notification Service applications that start and shut down only once can use the `proxy_id` to determine if their subscription has been cancelled automatically or by the system administrator.

Return Value

This operation returns the new proxy's object reference. The new `proxy_id` is also returned through the `proxy_id` out parameter.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating a Subscription" on page -10.

C++ code example:

```
CosNotifyChannelAdmin::ProxySupplier_var generic_proxy =
       consumer_admin->obtain_notification_push_supplier(
          CosNotifyChannelAdmin::STRUCTURED_EVENT,
          proxy_id
        );
```

```
CosNotifyChannelAdmin::StructuredProxyPushSupplier_var proxy =
      CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(
            generic_proxy.in ()
          );
```

CosNotifyChannelAdmin::ConsumerAdmin::get_proxy_supplier

Synopsis

Returns the proxy push supplier object created using the consumer admin object `obtain_notification_push_supplier` operation.

OMG IDL

```
ProxySupplier get_proxy_supplier (
       in ProxyID proxy_id )
    raises ( ProxyNotFound );
```

Exceptions

`CosNotifyChannelAdmin::``ProxyNotFound`
Indicates that the `ProxyID` could not be found.

Descriptions

Used when unsubscribing. This operation is used in subscriber applications to return the proxy push supplier object created using the consumer admin object `obtain_notification_push_supplier` operation. The `ProxyID` input parameter uniquely identifies the proxy object. Callers should be aware that the proxy object can be destroyed either due to an error in delivering a transient subscription or through an `ntsadmin` administrative command. When a proxy object is destroyed, the `ProxyID` associated with it is invalidated. If the `ProxyID` is invalid, a `ProxyNotFound` exception is raised. The subscriber must narrow the proxy supplier to `CosNotifyChannelAdmin::StructuredProxyPushSupplier`.

Return Value

Returns the object reference for the existing proxy.

Examples

C++ code example:

```
CosNotifyChannelAdmin::ProxySupplier_var generic_proxy =
        m_consumer_admin->get_proxy_supplier(
```

```
          m_subscription_info.news_proxy_id()
        );

CosNotifyChannelAdmin::StructuredProxyPushSupplier_var proxy =
      CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(
          generic_proxy.in()
      );
```

CosNotifyChannelAdmin::SupplierAdmin Class

This class is used by event poster applications. The OMG IDL for this class is as follows:

```
Module CosNotifyChannelAdmin
{
 interface SupplierAdmin :
        CosNotification::QoSAdmin,
        CosNotifyComm::NotifyPublish,
        CosNotifyFilter::FilterAdmin,
        CosEventChannelAdmin::SupplierAdmin {

     ProxyConsumer obtain_notification_push_consumer (
               in ClientType ctype,
               out ProxyID proxy_id)
        raises ( AdminLimitExceeded );
  };
}; //SupplierAdmin
```

CosNotifyChannelAdmin::SupplierAdmin:: obtain_notification_push_consumer

Synopsis

Creates proxy push consumer objects.

OMG IDL

```
ProxyConsumer obtain_notification_push_consumer (
             in ClientType ctype,
             out ProxyID proxy_id)
        raises ( AdminLimitExceeded );
```

Exceptions

`CosNotifyChannelAdmin::AdminLimitExceeded`
Never raised.

`CORBA::IMP_LIMIT`
Indicates the following problem:
`Tobj_Notification::SUB_UNSUPPORTED_CLIENT_TYPE`

Description

Used when posting events. This operation is used in poster applications to create proxy push consumer objects. `ClientType` must be set to `"CosNotifyChannelAdmin::STRUCTURED_EVENT"`. The `ProxyID` returned should be ignored. The Proxy Consumer must be narrowed the proxy supplier to `CosNotifyChannelAdmin::StructuredProxyPushConsumer`.

Note: Notification Service applications that start and shut down only once can use the `proxy_id` to determine if their subscription has been cancelled automatically or by the system administrator.

Return Value

This operation returns the new proxy's object reference. The new `proxy_id` is also returned through the `proxy_id` out parameter.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating and Posting Events" on page -3.

C++ code example:

```
CosNotifyChannelAdmin::ProxyConsumer_var generic_proxy_consumer =
      supplier_admin->obtain_notification_push_consumer(
        CosNotifyChannelAdmin::STRUCTURED_EVENT,
        proxy_id
      );

CosNotifyChannelAdmin::StructuredProxyPushConsumer_var
   proxy_push_consumer =
      CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(
        generic_proxy_consumer
      );
```

CosNotifyChannelAdmin::EventChannel Class

This class is used by event poster applications. The OMG IDL for this class is as follows:

```
Module CosNotifyChannelAdmin
{
 interface EventChannel :
       CosNotification::QoSAdmin,
       CosNotification::AdminPropertiesAdmin,
       CosEventChannelAdmin::EventChannel {

     readonly attribute ConsumerAdmin default_consumer_admin;
     readonly attribute SupplierAdmin default_supplier_admin;
     readonly attribute CosNotifyFilter::FilterFactory
                                default_filter_factory;
  };
}; //CosNotifyChannelAdmin
```

CosNotifyChannelAdmin::EventChannel:: ConsumerAdmin default_consumer_admin

Synopsis

Gets the ConsumerAdmin object.

OMG IDL

```
readonly attribute ConsumerAdmin default_consumer_admin;
```

Description

Used when subscribing and unsubscribing. This operation is used in subscriber applications to get the ConsumerAdmin object.

Return Value

Returns the object reference to the ConsumerAdmin object.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see “Getting the Event Channel, ConsumerAdmin Object, and Filter Factory Object” on page -8.

C++ code example:

```
channel->default_consumer_admin();
```

CosNotifyChannelAdmin::EventChannel:: ConsumerAdmin default_supplier_admin

Synopsis

Gets the SupplierAdmin object.

OMG IDL

```
readonly attribute SupplierAdmin default_supplier_admin;
```

Description

Used when posting events. This operation is used in event poster applications to get the SupplierAdmin object.

Return Value

SupplierAdmin object reference.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Creating and Posting Events" on page -3.

C++ code example:

```
channel->default_supplier_admin();
```

CosNotifyChannelAdmin::EventChannel::default_filter_factory

Synopsis

Gets the default FilterFactory object.

OMG IDL

```
readonly attribute CosNotifyFilter::FilterFactory
            default_filter_factory;
```

Description

Used when subscribing. This operation is used in subscriber applications to get the default FilterFactory object.

Return Value

Default FilterFactory object reference.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Getting the Event Channel, ConsumerAdmin Object, and Filter Factory Object" on page -8.

C++ code example:

```
channel->default_filter_factory();
```

CosNotifyChannelAdmin::EventChannelFactory Class

This class is used by event poster applications. The OMG IDL for this class is as follows:

```
Module CosNotifyChannelAdmin
{
 interface EventChannelFactory {
     EventChannel get_event_channel ( in ChannelID id )
     raises (ChannelNotFound);
   };
}; //CosNotifyChannelAdmin
```

CosNotifyChannelAdmin::EventChannelFactory::get_event_channel

Synopsis

Gets the `EventChannel` object.

OMG IDL

```
EventChannel get_event_channel ( in ChannelID id )
     raises (ChannelNotFound);
```

Exceptions

`CosNotifyChannelAdmin::ChannelNotFound`
Indicates the channel cannot be found.

Description

Used when subscribing, unsubscribing, and posting events. This operation is used in applications to get the `EventChannel` object. When subscribing, the EventChannel object is used to get the filter factory object and the ConsumerAdmin object. When unsubscribing, the EventChannel object is used to get the ConsumerAdmin object.When posting an event, the EventChannel object is used to get the SupplierAdmin object. The `ChannelID` parameter that is passed in must be set to `Tobj_Notification::DEFAULT_CHANNEL`; otherwise, the `ChannelNotFound` exception is raised.

Return Value

Returns the default event channel's object reference.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Getting the Event Channel" on page -2 and "Getting the Event Channel, ConsumerAdmin Object, and Filter Factory Object" on page -8.

C++ code example:

```
channel_factory->get_event_channel(
    Tobj_Notification::DEFAULT_CHANNEL );
```

CosNotifyComm::StructuredPushConsumer Interface

This interface is used by event subscriber applications for event delivery. You must implement this interface so that the Notification Service can invoke on it to deliver events to subscribers. It has three methods which you have to implement.

The OMG IDL for this class is as follows:

```
Module CosNotifyComm
{
 interface StructuredPushConsumer : NotifyPublish {

    void push_structured_event(
          in CosNotification::StructuredEvent event)
      raises(CosEventComm::Disconnected);
    void disconnect_structured_push_consumer:
    //The following operations are inherited.
    void offer_change(
          in CosNotification::EventTypeSeq added,
```

```
            in CosNotification::EventTypeSeq removed )
        raises ( InvalidEventType );
    };
}; //CosNotifyComm
```

CosNotifyComm::StructuredPushConsumer::push_structured_event

Synopsis

Delivers a structured event.

OMG IDL

```
void push_structured_event(
            in CosNotification::StructuredEvent event)
        raises(CosEventComm::Disconnected);
```

Exceptions

`CosEventComm::`**Disconnected**
The subscriber should never raise this exception.

Description

Used when subscribing. This operation is implemented by the subscriber's callback object and is invoked by the Notification Service each time a structured event is delivered. This operation contains a single input parameter, which is a structured event.

Note: This operation will not be called in a transaction. Also, when this operation is called, it must return quickly because the Notification Service might not start delivering events to other subscribers until this operation returns.

Examples

Note: Code examples shown here are abbreviated. For complete code examples, see "Implementing the CosNotifyComm::StructuredPushConsumer Interface" on page -5.

C++ code example:

```
virtual void push_structured_event(
        const CosNotification::StructuredEvent& notification );
         {
         // Process the event.
         }
```

CosNotifyComm::StructuredPushConsumer:: disconnect_structured_push_consumer

Synopsis

Never invoked.

OMG IDL

```
void disconnect_structured_push_consumer;
```

Description

This operation is never invoked. The subscriber application must provide a stubbed-out version of this operation.

Examples

C++ code example:

```
virtual void push_structured_event(
      const CosNotification::StructuredEvent& notification );
  {
    throw new CORBA::NO_IMPLEMENT();
  }
```

CosNotifyComm::StructuredPushConsumer::Offer_change

Synopsis

Never invoked.

OMG IDL

```
 void offer_change(
          in CosNotification::EventTypeSeq added,
          in CosNotification::EventTypeSeq removed )
      raises ( InvalidEventType );
```

Exceptions

`CosNotifyComm::InvalidEventType`

The subscriber should never raise this exception.

Description

This operation is never invoked. The subscriber application must provide a stubbed-out version of this operation.

Examples

C++ code example:

```
virtual void offer_change(
      const CosNotification::EventTypeSeq& added,
      const CosNotification::EventTypeSeq& removed )
    {
      throw CORBA::NO_IMPLEMENT();
    }
```

Exception Minor Codes

This section provides information about the Notification Service exception symbols and minor codes. The minor codes are in the `Tobj_Events.idl` and `Tobj_Notification.idl` files. These files are located in the `tuxdir\include` directory (for Microsoft Windows systems) and `tuxdir/include` directory (for UNIX systems).

Table 2-4 and Table 2-5 list the exception symbols and corresponding minor codes for the Tobj_Events and Tobj_Notification exceptions respectively. CORBA system events have a minor code field and those minor codes are also defined in these tables.

Note: The exception symbols are organized within the tables by the higher-level exceptions (`CORBA::IMP_LIMIT`, `CORBA::CORBA::BAD_PARAM`, `CORBA::BAD_INV_ORDER`, `CORBA::INV_OBHJREF`, and `CORBA::OBJECT_NOT_EXIST`) and listed in alphabetical order.

Table 2-4 Tobj_Events Exception Minor Codes

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
CORBA::IMP_LIMIT Exceptions		
Tobj_Events:: POST_DOMAIN_AND_TYPE_TOO_LONG This exception is raised by: • Tobj_SimpleEvents::Channel:: push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user specified a domain name and type name whose combined length was greater than 31 characters.	5455580D
Tobj_Events:: POST_DOMAIN_CONTAINS_SEPARATOR This exception is raised by: • Tobj_SimpleEvents::Channel:: push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user specified a domain name that contained the "." character.	54555802
Tobj_Events::POST_EMPTY_DOMAIN This exception is raised by: • Tobj_SimpleEvents::Channel:: push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user specified an empty domain name.	5455580B
Tobj_Events::POST_EMPTY_TYPE This exception is raised by: • Tobj_SimpleEvents::Channel:: push_structured_event • CosNotifyChannelAdmin:: StructuredProxyPushConsumer:: push_structured_event	When posting an event, the user specified an empty type name.	5455580C

Table 2-4 Tobj_Events Exception Minor Codes (Continued)

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
`Tobj_Events::POST_SYSTEM_EVENTS_UNSUPPORTED` This exception is raised by: • `Tobj_SimpleEvents::Channel::push_structured_event` • `CosNotifyChannelAdmin::StructuredProxyPushConsumer::push_structured_event`	When posting an event, the user tried to post an Oracle Tuxedo system event; that is, the domain name is `"TMEVT"` and the type name starts with the `"."` character.	54555804
`Tobj_Events::POST_TYPE_CONTAINS_SEPARATOR` This exception is raised by: • `Tobj_SimpleEvents::Channel::push_structured_event` • `CosNotifyChannelAdmin::StructuredProxyPushConsumer::push_structured_event`	When posting an event, the user specified a type name that contained the `"."` character.	54555803
`Tobj_Events::POST_UNSUPPORTED_PRIORITY_VALUE` This is exception is raised by: • `Tobj_SimpleEvents::Channel::push_structured_event` • `CosNotifyChannelAdmin::StructuredProxyPushConsumer::push_structured_event`	When posting an event, the user added a `"Priority"` field in the variable header. However, the user did not set the field's value to a `"short"` in the range of 1–100.	54555801
`Tobj_Events::POST_UNSUPPORTED_VALUE_IN_ANY` This exception is raised by: • `Tobj_SimpleEvents::Channel::push_structured_event` • `CosNotifyChannelAdmin::StructuredProxyPushConsumer::push_structured_event`	When posting an event, the user put an unsupported type (for example, a structure, union, sequence, etc.) into one of the "anys" in the structured event field. The unsupported type is in the variable header's value field, the filterable data's value field, or the remainder_of_body field.	54555800

Table 2-4 Tobj_Events Exception Minor Codes (Continued)

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
`Tobj_Events::SUB_DOMAIN_AND_TYPE_TOO_LONG` This exception is raised by: • `Tobj_SimpleEvents::Channel::subscribe` • `CosNotifyChannelAdmin::StructuredProxyPushSupplier::connect_structured_push_consumer`	When subscribing, the user specified a domain name and type name whose combined length is greater than 255 characters.	54555809
`Tobj_Events::SUB_DOMAIN_BEGINS_WITH_SYSEV` This exception is raised by: • `Tobj_SimpleEvents::Channel::subscribe` • `CosNotifyFilter::Filter::add_constraints`	When subscribing, the user specified a domain name that begins with the `"."` character.	54555805
`Tobj_Events::SUB_EMPTY_DOMAIN` This exception is raised by: • `Tobj_SimpleEvents::Channel::subscribe` • `CosNotifyFilter::Filter::add_constraints`	The user specified an empty domain name when subscribing.	54555807
`Tobj_Events::SUB_EMPTY_TYPE` This exception is raised by: • `Tobj_SimpleEvents::Channel::subscribe` • `CosNotifyFilter::Filter::add_constraints`	The user specified an empty type name when subscribing.	54555808
`Tobj_Events::SUB_FILTER_TOO_LONG` This exception is raised by: • `Tobj_SimpleEvents::Channel::subscribe` • `CosNotifyFilter::Filter::add_constraints`	The user specified a data filter expression longer than 255 characters.	5455580A

Table 2-4 Tobj_Events Exception Minor Codes (Continued)

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
`Tobj_Events::SUB_NAME_TO_LONG` This exception is raised by: • `Tobj_SimpleEvents::Channel::push_structured_event` • `CosNotifyChannelAdmin::StructuredProxyPushConsumer::push_structured_event`	When subscribing, the user specified a subscription name longer than 127 characters.	5455580E
`Tobj_Events::TRANSIENT_ONLY_CONFIGURATION` This exception is raised by: • `Tobj_SimpleEvents::Channel::subscribe` • `CosNotifyChannelAdmin::StructuredProxyPushSupplier::connect_structured_push_consumer`	The user tried to create a persistent subscription, but the system was configured to support transient subscriptions only.	54555806
`CORBA::BAD_PARAM` Exceptions		
`Tobj_Events::INVALID_CHANNEL_ID` This exception is raised by: • `Tobj_SimpleEvents::ChannelFactory::find_channel`	When looking up the channel using the Simple Events API, the user specified an invalid channel ID, that is, a channel ID that is not `Tobj_SimpleEvents::DEFAULT_CHANNEL`.	54555813

Table 2-4 Tobj_Events Exception Minor Codes (Continued)

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
`Tobj_Events::INVALID_SUBSCRIPTION_ID` This exception is raised by: • `Tobj_SimpleEvents::Channel::unsubscribe` • `CosNotifyChannelAdmin::ConsumerAdmin::get_proxy_supplier` • `Tobj_SimpleEvents::Channel::exists`	When unsubscribing using the Simple Events API, the user specified an invalid subscription ID, that is, a non-existent or a CosNotification subscription ID. When looking up a subscription using the CosNotification Service API, the user specified an invalid subscription ID, that is, a non-existent or a Simple Events API subscription ID. When calling the `exists` operation using the Oracle Simple Events API, the user passed in a CosNotification `subscription_id`.	54555812
`Tobj_Events::SUB_INVALID_FILTER_EXPRESSION` This exception is raised by: • `Tobj_SimpleEvents::Channel::subscribe` • `CosNotifyFilter::Filter::add_constraints`	When subscribing, the user specified an invalid data filter expression. This either means that there is a syntax error in the expression or that one of the field names in the expression is not defined as an FML field. Check that you have correctly created FML field tables that contain all fields that you want to data filter on, and check that the `UBBCONFIG` file is properly configured so that the field table files can be found.	54555810

Table 2-4 Tobj_Events Exception Minor Codes (Continued)

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
`Tobj_Events::SUB_UNSUPPORTED_QOS_VALUE` This exception is raised by: • `Tobj_SimpleEvents::Channel::subscribe` • `CosNotifyChannelAdmin::StructuredProxyPushSupplier::set_qos`	**54555811** When subscribing, the user specified an invalid subscription quality of service. For the Simple Events API, this means that the quality of service specified did not meet one of the following requirements: • The sequence must be of length one. • The name must be `Tobj_SimpleEvents::SUBSCRIPTION_TYPE`. • The value must be either `Tobj_SimpleEvents::TRANSIENT_SUBSCRIPTION` or `Tobj_SimpleEvents::PERSISTENT_SUBSCRIPTION`. For the CosNotification Service API, this means that the quality of service specified did not meet one of the following requirements: • The quality of service must contain a name/value pair where the name is `Tobj_Notification::SUBSCRIPTION_TYPE` and the value is `Tobj_Notification::TRANSIENT_SUBSCRIPTION` or `Tobj_Notification::PERSISTENT_SUBSCRIPTION`. • The quality of service may contain a name/value pair where the name is `Tobj_Notification::SUBSCRIPTION_NAME` and the value is a string containing the subscription's administrative name.	

Table 2-4 Tobj_Events Exception Minor Codes (Continued)

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
CORBA::INV_OBHJREF		
`Tobj_Events::SUB_NIL_CALLBACK_REF` This exception is raised by: • `Tobj_SimpleEvents::Channel::subscribe` • `CosNotifyChannelAdmin::StructuredProxyPushSupplier::connect_structured_push_consumer`	When subscribing, the user specified a NIL object reference for the callback object which receives events.	54555830

Table 2-5 Tobj_Notification Exception Minor Codes

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
CORBA::IMP_LIMIT Exceptions		
Tobj_Notification::SUB_ADD_CONS_ON_TIMED_OUT_FILTER This exception is raised by: • `CosNotifyFilter::Filter::add_constraints`	A CosNotification subscriber waited more than five minutes after creating a filter to call `add_constraints` on the filter. This means that the filter has been destroyed (timed out) and the subscriber must create a new filter.	54555858
Tobj_Notification::SUB_ADD_CONS_TO_ADDED_FILTER This exception is raised by: • `CosNotifyFilter::Filter::add_constraints`	A CosNotification subscriber called `add_constraints` on a filter that had already been added to a proxy.	5455585E

Table 2-5 Tobj_Notification Exception Minor Codes (Continued)

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
Tobj_Notification:: SUB_ADDED_TIMED_OUT_FILTER This exception is raised by: • `CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: add_filter`	After creating a filter and calling "add_constraints" on it, a CosNotification subscriber waited more than five minutes to call `add_filter` to add the filter to the proxy. This means that the filter has been destroyed (timed out) and that the subscriber must create a new filter.	5455585D
Tobj_Notification:: SUB_ADD_FILTER_AFTER_CONNECT This exception is raised by: • `CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: add_filter`	A CosNotification subscriber called `add_filter` after connecting to the proxy.	54555852
Tobj_Notification:: SUB_CANT_SET_QOS_AFTER_CONNECT This exception is raised by: • `CosNotifyChannelAdmin::Structured ProxyPushSupplier::set_qos`	A CosNotification subscriber called `set_qos` after connecting to the proxy.	54555856
Tobj_Notification:: SUB_MULTIPLE_CALLS_TO_ADD_CONS This exception is raised by: • `CosNotifyFilter::Filter:: add_constraints`	A CosNotification subscriber called `add_constraints` more than once on a filter.	54555859
Tobj_Notification:: SUB_MULTIPLE_CALLS_TO_SET_FILTER This exception is raised by: • `CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: add_filter`	A CosNotification subscriber called `add_filter` more than once on a proxy.	54555851

Table 2-5 Tobj_Notification Exception Minor Codes (Continued)

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
Tobj_Notification::SUB_MULTIPLE_CALLS_TO_SET_QOS This exception is raised by: • `CosNotifyChannelAdmin::StructuredProxyPushSupplier::set_qos`	A CosNotification subscriber called `set_qos` more than once on a proxy.	54555855
Tobj_Notification::SUB_MULTIPLE_CONSTRAINTS_IN_LIST This exception is raised by: • `CosNotifyFilter::Filter::add_constraints`	When a CosNotification subscriber called `add_constraints` on a filter, the subscriber passed in a list of constraints that had more than one item; that is, the subscriber was trying to send in a list of data filters instead of one data filter.	5455585A
Tobj_Notification::SUB_MULTIPLE_TYPES_IN_CONSTRAINT This exception is raised by: • `CosNotifyFilter::Filter::add_constraints`	When a CosNotification subscriber called `add_constraints` on a filter, the subscriber passed on a constraint that had more than one domain/type set; that is, the subscriber was trying to send in a list of desired event types instead of one event type.	5455585B
Tobj_Notification::SUB_NIL_FILTER_REF This exception is raised by: • `CosNotifyChannelAdmin::StructuredProxyPushSupplier::add_filter`	A CosNotification subscriber passed a NIL filter object reference into `add_filter`.	54555853

Table 2-5 Tobj_Notification Exception Minor Codes (Continued)

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
Tobj_Notification:: SUB_NO_CUSTOM_FILTERS This exception is raised by: • `CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: add_filter`	A CosNotification subscriber passed a filter object that was not created by the default filter factory into `add_filter`. For example, a CosNotification subscriber implemented the `CosNotifyFilter::Filter` interface to do some kind of "custom" filtering and passed one of those filter objects into `add_filter`.	54555854
Tobj_Notification:: SUB_SET_FILTER_NOT_CALLED This exception is raised by: • `CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: connect_structured_push_ consumer`	A CosNotification subscriber did not call `add_filter` to the proxy before connecting to the proxy.	54555850
Tobj_Notification:: SUB_SET_QOS_NOT_CALLED This exception is raised by: • `CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: connect_structured_push_ consumer`	A CosNotification subscriber did not call `add_filter` to the proxy before connecting to the proxy.	54555857
Tobj_Notification:: SUB_SYSTEM_EVENTS_UNSUPPORTED This exception is raised by: • `CosNotifyChannelAdmin:: StructuredProxyPushSupplier:: set_qos`	A CosNotification subscriber passed in a domain name of `"TMEVT"` and a type name that begins with `"."`; that is, the CosNotification subscriber was trying to subscribe to Tuxedo system events. This is not supported. It is only supported by the Simple Events API.	5455585C

Table 2-5 Tobj_Notification Exception Minor Codes (Continued)

Exception Symbols	Definitions	Minor Codes (Hexadecimal)
Tobj_Notification:: SUB_UNSUPPORTED_CLIENT_TYPE This is exception raised by: • `ConsumerAdmin:: obtain_notification_push_ supplier` • `SupplierAdmin:: obtain_notification_push_ consumer`	When creating a proxy, a CosNotification subscriber or poster passed in a client type other than `CosNotifyChannelAdmin::ST RUCTURED_EVENT`.	5455585F
CORBA::OBJECT_NOT_EXIST Exception		
Tobj_Notification:: SUBSCRIPTION_DOESNT_EXIST This exception is raised by: • `StructuredProxyPushSupplier:: add_filter` • `StructuredProxyPushSupplier:: set_qos` • `StructuredProxyPushSupplier:: connect_structured_push_ consumer` • `StructuredProxyPushSupplier:: disconnect_structured_push_ supplier` **Note:** `connect_structured_push_ consumer` can raise this exception since a user can create the proxy, then use the `ntsadmin` utility to delete the subscription, and then call `connect_structured_push_ consumer` on the proxy.	A CosNotification subscriber called a method on a proxy that had already been destroyed. The proxy has been destroyed by one of the following actions: • The CosNotification subscriber disconnected the proxy. • The CosNotification subscriber waited more than five minutes from creating the proxy to connecting it; that is, it took longer than five minutes to complete the subscription. • The administrator used the `ntsadmin` utility to destroy the subscription.	54555880

CHAPTER 3

Using the Oracle Simple Events API

This chapter describes the development steps required to create Notification Service applications using the Oracle Simple Events API and the C++ languages.

This topic includes the following sections:

- Development Process
- Step 1: Writing an Application to Post Events
- Step 2: Writing an Application to Subscribe to Events
- Step 3: Compiling and Running Notification Service Applications

Note: The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Development Process

Table 3-1 outlines the development process for creating Notification Service applications.

Table 3-1 Development Process

Step	Description
1	Designing events
2	Writing an application that posts events
3	Writing an application that subscribes to events
4	Compiling a Notification Service application

These steps are explained in detail in subsequent topics.

Designing Events

The design of events is basic to any notification service. The design impacts not only the volume of information that is delivered to matching subscriptions, but the efficiency and performance of the Notification Service as well. Therefore, careful planning should be done to ensure that your Notification Service will be able to handle your needs now and allow for future growth. For a discussion of event design, see “Designing Events” on page -6.

Step 1: Writing an Application to Post Events

The following types of CORBA applications can post events:

- C++ clients, joint client/servers and servers.
- Foreign ORB clients.

To post events, an application must, at a minimum, implement the following functions:

- Get the event channel factory object reference and use it to get the event channel.
- Create and post events.

The following sections describe each of these functions.

Getting the Event Channel

Before the client application can post an event, it must first get the event channel.

This development step is illustrated in Listing 3-1. Listing 3-1 is based on the Notification Service sample applications that use the Oracle Simple Events API.

To get the event channel factory object reference, the `resolve_initial_references` method is invoked on the Bootstrap object using the `"Tobj_SimpleEventsService"` environmental object. The object reference is used to get the channel factory, which is in turn is used to get the event channel. Listing 3-1 show code examples in C++.

Listing 3-1 Getting the Event Channel (C++)

```
// Get the Simple Events channel factory object reference.
CORBA::Object_var channel_factory_oref =
  bootstrap.resolve_initial_references(
                      "Tobj_SimpleEventsService");
Tobj_SimpleEvents::ChannelFactory_var channel_factory =
  Tobj_SimpleEvents::ChannelFactory::_narrow(
                      channel_factory_oref.in());
// Use the channel factory to get the default channel.
Tobj_SimpleEvents::Channel_var channel =
  channel_factory->find_channel(
                      Tobj_SimpleEvents::DEFAULT_CHANNEL);
```

Creating and Posting Events

Before an event can be posted, it must be created. The following listings are based on the Notification Service sample applications.

Listing 3-2 show how this is implemented in C++. To report news to the events channel, this application executes the following steps:

1. Creates an event and sets the domain name and type name. In the code samples, the domain name is set to “News” and the event type is set to “Sports”.
2. Adds a field to the event’s filterable data to contain the story, sets the name of the added field to “Story”, and the value of the field to a string containing the story.
3. Uses the `push_structured_event` operation to post the event to the Notification Service.

Listing 3-2 Creating and Posting the Event (C++)

```
// Create an event.
CosNotification::StructuredEvent notification;
// Set the domain to "News".
notification.header.fixed_header.event_type.domain_name =
    CORBA::string_dup("News");
// Set the type to the news category.
notification.header.fixed_header.event_type.type_name =
    CORBA::string_dup("Sports");
// Add one field, which will contain the story, to the
// event's filterable data. Set the field's name to
// "Story" and value to a string containing the story.
notification.filterable_data.length(1);
notification.filterable_data[0].name =
    CORBA::string_dup("Story");
notification.filterable_data[0].value <<= "John Smith wins again";
// Post the event.
// Subscribers who subscribed to events whose domain is
// "News" and whose type matches the news category will
// receive this event
channel->push_structured_event(notification);
```

Step 2: Writing an Application to Subscribe to Events

The following types of CORBA applications can subscribe to events:

- C++ joint client/servers and servers.
- Foreign ORB clients.

To subscribe to events, an application must, at a minimum, implement the following functions:

- Implement a CosNotifyComm OMG IDL interface that supports the `push_structured_event` operation.
- Get the event channel factory object reference and use it to get the event channel.
- Define and create a subscription that includes the callback object reference.

- Create a callback object that implements the CosNotifyComm::StructuredPushConsumer interface.

Implementing the CosNotifyComm::StructuredPushConsumer Interface

In order for the callback object to receive events, it must implement the CosNotifyComm::StructuredPushConsumer interface that supports the `push_structured_event` operation. When an event occurs that has a matching subscription, the Notification Service invokes this operation on the callback object to push the event to the subscriber application.

The CosNotifyComm::StructuredPushConsumer interface also defines the operations `offer_change` and `disconnect_structured_push_consumer`. The Notification Service never invokes these operations, so you should implement stubbed out versions that throw `CORBA::NO_IMPLEMENT`.

Listing 3-3 and Listing 3-4 show how this interface is implemented in C++.

Listing 3-3 Sample CosNotifyComm::StructuredPushConsumer Interface Implementation (NewsConsumer_i.h)

```
#ifndef _news_consumer_i_h
#define _news_consumer_i_h
#include "CosNotifyComm_s.h"
// For the servant class to receive news events,
// it must implement the CosNotifyComm::StructuredPushConsumer
// idl interface.
class NewsConsumer_i : public POA_CosNotifyComm::StructuredPushConsumer
{
  public:
    // This method will be called when a news event occurs.
    virtual void push_structured_event(
      const CosNotification::StructuredEvent& notification
    );
    // OMG's CosNotifyComm::StructuredPushConsumer idl
    // interface defines the methods "offer_change" and
    // "disconnect_structured_push_consumer". Since the
```

```
    // Notification Service never invokes these methods, just
    // have them throw a CORBA::NO_IMPLEMENT exception
    virtual void offer_change(
      const CosNotification::EventTypeSeq& added,
      const CosNotification::EventTypeSeq& removed )
    {
      throw CORBA::NO_IMPLEMENT();
    }
    virtual void disconnect_structured_push_consumer()
    {
      throw CORBA::NO_IMPLEMENT();
    }
};
#endif
```

Listing 3-4 Sample CosNotifyComm::StructuredPushConsumer Interface Implementation (NewsConsumer_i.cpp)

```
#include "NewsConsumer_i.h"
#include <iostream.h>
//-------------------------------------------------------------
// Subscriber.cpp creates a simple events subscription to "News"
// events and has the events delivered to a NewsConsumer_i
// object. When a news event occurs (this happens when a user
// runs the Reporter application and reports a news story), this
// method will be invoked:
void NewsConsumer_i::push_structured_event(
  const CosNotification::StructuredEvent& notification )
  {
// Extract the story from the first field in the event's
// filterable data.
char* story;
notification.filterable_data[0].value >>= story;
// For coding simplicity, assume "story" is not "null".
// Print out the event.
cout
```

```
    << "-----------------------------------------------------"
    << endl
    << "Category : "
    << notification.header.fixed_header.
                          event_type.type_name.in()
    << endl
    << "Story    : "
    << story
    << endl;
...
}
```

Getting the Event Channel

This step is the same for event posters and event subscribers. For a discussion of this step, see “Implementing the CosNotifyComm::StructuredPushConsumer Interface” on page -5.

Creating a Callback Object

To receive events, the application must also be a server; that is, the application must implement a callback object that can be invoked (called back) when an event occurs that matches the subscriber’s subscription.

Creating a callback object includes the following steps:

Note: The following steps apply to an Oracle Tuxedo CORBA joint client/server. Oracle Tuxedo CORBA servers can also subscribe to events.

1. Create a callback object. Callback objects can be implemented using either the BEAWrapper Callback API or the CORBA Portable Object Adaptor (POA).
2. Create the servant.
3. Create an object reference to the callback servant.

For a complete description of the BEAWrapper Callbacks object and its methods, see the Joint Client/Servers chapter in the *CORBA Programming Reference*.

pay attention to”BEAWrapper”

Note: Using the BEAWrapper Callback object to create a callback object is discussed below. For a discussion of how to implement a callback object using the POA, see *Using CORBA Server-to-Server Communication*.

Listing 3-5 show show to use the BEAWrapper Callbacks object to create a callback object in C++. In the code examples, the `NewsConsumber_i servant` is created and the `start_transient` method is used to create a transient object reference.

Listing 3-5 Sample Code for Creating a Callback Object With Transient Object Reference (Introductory Application Subscriber.cpp)

```
// Create a callback wrapper object since this client needs to
// support callbacks.
BEAWrapper::Callbacks wrapper(orb.in());
NewsConsumer_i* news_consumer_impl = new NewsConsumer_i;
CORBA::Object_var news_consumer_oref =
      wrapper.start_transient(
        news_consumer_impl,
        CosNotifyComm::_tc_StructuredPushConsumer->id()
      );
CosNotifyComm::StructuredPushConsumer_var
    news_consumer =
        CosNotifyComm::StructuredPushConsumer::_narrow(
        news_consumer_oref.in()
      );
```

Creating a Subscription

In order for the subscriber to receive events, it must subscribe to the Notification Service. You can create either a transient subscription or a persistent subscription.

Listing 3-6 from the Introductory sample application, show how to create a transient subscription in C++.

The following steps must be performed:

1. Set the subscription's quality of service (QoS) to either transient or persistent.

2. Determine the `subscription_name` (optional), `domain_name`, `type_name`, and `data_filter` (optional).
3. Create the subscription. The subscription sets the `domain_name`, `type_name`, and `data_filter` (optional), the Quality of Service (QoS), and supplies the object reference to the subscriber's callback object to the Notification Service.

Listing 3-6 Creating a Transient Subscription (C++)

```
// Set the quality of service to TRANSIENT.
CosNotification::QoSProperties qos;
qos.length(1);
qos[0].name =
   CORBA::string_dup(Tobj_SimpleEvents::SUBSCRIPTION_TYPE);
qos[0].value <<=
   Tobj_SimpleEvents::TRANSIENT_SUBSCRIPTION;
// Set the type to the news category.
const char* type = "Sports";
// Create the subscription. Set the domain to "News" and
// the data filter to age greater than 30.
Tobj_SimpleEvents::SubscriptionID subscription_id =
  channel->subscribe(
    subscription_name,
    "News",  // domain
   "Sports", // type
    "Age > 30",  // Data filter.
    qos,
    news_consumer.in()
  );
```

Note: When you use data filtering, you must also perform some configuration tasks. For a discussion of data filtering configuration requirements, see "Configuring Data Filters" on page -2.

Listing 3-7, which show code in the Advanced sample application in C++, illustrates the coding steps required to create a persistent subscription to the Notification Service. The steps required to

create a persistent subscription are the same as those required to create a transient subscription, as described previously.

Note: While the code examples shown here assume that the `news_consumer` callback object has a persistent object reference, you can also create persistent subscriptions with transient callback object references. For a discussion of transient versus persistent callback object references, see Table 2-3.

Listing 3-7 Creating a Persistent Subscription (Advanced Subscriber.cpp)

```
CosNotification::QoSProperties qos;
qos.length(1);
qos[0].name =
  CORBA::string_dup(Tobj_SimpleEvents::SUBSCRIPTION_TYPE);
qos[0].value <<= Tobj_SimpleEvents::PERSISTENT_SUBSCRIPTION;
CosNotifyComm::StructuredPushConsumer_var
  news_consumer =
    CosNotifyComm::StructuredPushConsumer::_narrow(
        news_consumer_oref.in()
      );
Tobj_SimpleEvents::SubscriptionID sub_id =
 channel->subscribe(
    subscription_info.subscription_name(),
    "News", // domain
    "Sports", // type
    "",     // No data filter.
    qos,
    news_consumer.in()
  )
);
```

Threading Considerations for C++ Joint Client/Server Applications

A joint client/server application may first function as a client application and then switch to functioning as a server application. To do this, the joint client/server application turns complete control of the thread to the Object Request Broker (ORB) by making the following invocation:

```
orb -> run();
```

If a method in the server portion of a joint client/server application invokes `ORB::shutdown()`, all server activity stops and control is returned to the statement after `ORB::run()` is invoked in the server portion of the joint client/server application. Only under this condition does control return to the client functionality of the joint client/server application.

Since a client application has only a single thread, the client functionality of the joint client/server application must share the central processing unit (CPU) with the server functionality of the joint client/server application. This sharing is accomplished by occasionally checking with the ORB to see if the joint client/server application has server application work to perform. Use the following code to perform the check with the ORB:

```
if ( orb->work_pending() ) orb->perform_work();
```

After the ORB completes the server application work, the ORB returns to the joint client/server application, which then performs client application functions. The joint client/server application must remember to occasionally check with the ORB; otherwise, the joint client/server application will never process any invocations.

You should be aware that the ORB cannot service callbacks while the joint client/server application is blocking on a request. If a joint client/server application invokes an object in another Oracle Tuxedo CORBA server application, the ORB blocks while it waits for the response. While the ORB is blocking, it cannot service any callbacks, so the callbacks are queued until the request is completed.

Step 3: Compiling and Running Notification Service Applications

The final step in the development of a Notification Service application is to compile, build, and run the application. To do this, you need to perform the following steps.

1. Generate the required client stub and skeleton files to define interfaces between the Notification Service and event poster and subscriber applications. Event poster applications can be clients, joint client/servers, or servers. Event subscriber applications can be joint client/servers or servers.
2. Compile the application code and link against the skeleton and client stub files.
3. Build the application.
4. Run the application.

Generating the Client Stub and Skeleton Files

To generate the client stub and skeleton files, you must execute the `idl` command for each of the Notification IDL files that your application uses. Table 3-2 shows the `idl` commands used for each type of subscriber.

Table 3-2 idl Command Requirements

Language	Oracle Tuxedo CORBA Joint Client/Server	Oracle Tuxedo CORBA Server
C++	`idl -P`	`idl`

The following is an example of an `idl` command:

```
>idl -IC:\tuxdir\include C:\tuxdir\include\CosEventComm.idl
```

Table 3-3 lists the IDL files required by each type of Notification Service application that uses the Oracle Simple Events Interface.

Table 3-3 IDL Files Required by Notification Service Applications

Application Type	Required OMG IDL Files

Event poster (can be a client, a joint client/server, or a server). (Stubs are required for all files.)	`CosEventComm.idl`
	`CosNotification.idl`
	`CosNotifyComm.idl`
	`Tobj_Events.idl`
	`Tobj_SimpleEvents.idl`

Subscriber (can be a server or a joint client/server). (Stubs are required for all files. Skeleton is required for the `CosNotifyComm.idl` file.)	`CosEventComm.idl`
	`CosNotification.idl`
	`CosNotifyComm.idl`
	`Tobj_Events.idl`
	`Tobj_SimpleEvents.idl`

Building and Running Applications

The build procedure differs depending on the type of Notification Service application you are building. Table 3-4 provides an overview of the commands and types of files used to build each type of the Notification Service application.

Table 3-4 Application Build Requirements

Application Type	Client	Joint Client/Server	Server
C++ Events Poster	Use the `buildobjclient` command to compile the application files and the IDL stubs.	Use the `buildobjclient` command with the `-P` option to compile the application files and the IDL stubs.	Use the `buildobjserver` command to compile the application files and the IDL client stubs.
C++ Events Subscriber	Not applicable.	Use the `buildobjclient` command with the `-P` option to compile the application files, the IDL stubs, the IDL skeletons, and link with the BEAWrapper library.	Use the `buildobjserver` command to compile the application files, the IDL stubs, and the IDL skeletons.

Listing 3-8 shows the commands used for a C++ poster application (`Reporter.cpp`) on a Microsoft Windows system. To form a C++ executable, the `idl` command is run on the required IDL file and the `buildobjclient` command compiles the C++ client application file and the IDL stubs.

Listing 3-8 C++ Reporter Application Build and Run Commands (Microsoft Windows)

```
# Run the idl command.
idl -IC:\tuxdir\include C:\tuxdir\include\CosEventComm.idl \
C:\tuxdir\include\CosNotification.idl \
C:\tuxdir\include\CosNotifyComm.idl \
C:\tuxdir\include\Tobj_Events.idl \
C:\tuxdir\include\Tobj_SimpleEvents.idl
# Run the buildobjclient command.
buildobjclient -v -o subscriber.exe -f " \
    -DWIN32                             \
    Reporter.cpp                        \
    CosEventComm_c.cpp                  \
    CosNotification_c.cpp               \
    CosNotifyComm_c.cpp                 \
    Tobj_Events_c.cpp                   \
```

```
    Tobj_SimpleEvents_c.cpp                \
    "
# Run the application.
is_reporter
```

Listing 3-9 and Listing 3-10 show the commands used for a C++ subscriber application (`Subscriber.cpp`) on Microsoft Windows and UNIX respectively. To form a C++ executable, the `buildobjclient` command, with the `-P` option, compiles the joint client/server application files (`Subscriber.cpp` and `NewsConsumer_i.cpp`), the IDL stubs, and the IDL skeleton (`CosNotifyComm_s.cpp`).

Listing 3-9 C++ Subscriber Application Build and Run Commands (Microsoft Windows)

```
# Run the idl command.
idl -P -IC:\tuxdir\include C:\tuxdir\include\CosEventComm.idl \
C:\tuxdir\include\CosNotification.idl \
C:\tuxdir\include\CosNotifyComm.idl \
C:\tuxdir\include\Tobj_Events.idl \
C:\tuxdir\include\Tobj_SimpleEvents.idl
# Run the buildobjclient command.
buildobjclient -v -P -o subscriber.exe -f "  \
  -DWIN32                                    \
  Subscriber.cpp                             \
  NewsConsumer_i.cpp                         \
  CosEventComm_c.cpp                         \
  CosNotification_c.cpp                      \
  CosNotifyComm_c.cpp                        \
  CosNotifyComm_s.cpp                        \
  Tobj_Events_c.cpp                          \
  Tobj_SimpleEvents_c.cpp                      \
c:\tuxdir\lib\libbeawrapper.lib              \
"
# Run the application.
is_subscriber
```

Listing 3-10 C++ Subscriber Application Build and Run Commands (UNIX)

```
# Run the idl command.
idl -P -I/usr/local/tuxdir/include
/usr/local/tuxdir/include/CosEventComm.idl \
/usr/local/tuxdir/include/CosNotification.idl \
/usr/local/tuxdir/include/CosNotifyComm.idl \
/usr/local/tuxdir/include/Tobj_Events.idl \
/usr/local/tuxdir/include/Tobj_SimpleEvents.idl
# Run the buildobjclient command.
buildobjclient -v -P -o subscriber -f "     \
  Subscriber.cpp                            \
  NewsConsumer_i.cpp                        \
  CosEventComm_c.cpp                        \
  CosNotification_c.cpp                     \
  CosNotifyComm_c.cpp                       \
  CosNotifyComm_s.cpp                       \
  Tobj_Events_c.cpp                         \
  Tobj_SimpleEvents_c.cpp                   \
  -lbeawrapper                              \
"
# Run the application.
is_subscriber
```

CHAPTER 4

Using the CosNotification Service API

This chapter describes the development steps required to create Notification Service applications using the CosNotification Service API and the C++ programming language.

This topic includes the following sections:

- Development Process
- Step 1: Writing an Application to Post Events
- Step 2: Writing an Application to Subscribe to Events
- Step 3: Compiling and Running Notification Service Applications

Note: The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Development Process

Table 4-1 outlines the development process for creating Notification Service applications.

Table 4-1 Development Process

Step	Description
1	Designing events
2	Writing an application that posts events
3	Writing an application that subscribes to events
4	Compiling a Notification Service application

These steps are explained in detail in subsequent topics.

Designing Events

The design of events is basic to any notification service. The design impacts not only the volume of information that is delivered to matching subscriptions, but the efficiency and performance of the Notification Service as well. Therefore, careful planning should be done to ensure that your Notification Service will be able to handle your needs now and allow for future growth. For a discussion of event design, see “Designing Events” on page -6.

Step 1: Writing an Application to Post Events

The following types of CORBA applications can post events:

- C++ clients, joint client/servers and servers.
- Foreign ORB clients.

To post events, an application must, at a minimum, implement the following functions:

- Get the event channel factory object reference and use it to get the event channel.
- Create and post events.

The following sections describe each of these functions.

Getting the Event Channel

Before the client application can post an event, it must get the event channel.

This development step is illustrated in Listing 4-1. Listing 4-1 is code from the `Reporter.cpp` file in the Introductory sample application that uses the CosNotification Service API.

To get the event channel factory object reference, the `resolve_initial_references` method is invoked on the Bootstrap object using the `"NotificationService"` environmental object. The object reference is used to get the channel factory, which is, in turn, is used to get the event channel. Listing 4-1 shows code examples in C++.

Listing 4-1 Getting the Event Channel (Reporter.cpp)

```
// Get the CosNotification channel factory object reference.
CORBA::Object_var channel_factory_oref =
       bootstrap.resolve_initial_references(
        "NotificationService" );
CosNotifyChannelAdmin::EventChannelFactory_var
   channel_factory =
       CosNotifyChannelAdmin::EventChannelFactory::_narrow(
          channel_factory_oref.in() );
// use the channel factory to get the default channel
CosNotifyChannelAdmin::EventChannel_var channel =
   channel_factory->get_event_channel(
       Tobj_Notification::DEFAULT_CHANNEL );
```

Creating and Posting Events

To post events, you must get the SupplierAdmin object, use it to create a proxy, create the event, and then post the event to the proxy.

Listing 4-2 shows how this is implemented in C++.

Listing 4-2 Creating and Posting the Event (Reporter.cpp)

```
// Since we are a supplier (that is, we post events),
// get the SupplierAdmin object
```

```
CosNotifyChannelAdmin::SupplierAdmin_var supplier_admin =
       channel->default_supplier_admin();
// Use the supplier admin to create a proxy. Events are posted
// to the proxy (unlike the simple events interface where events
// are posted to the channel).
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxyConsumer_var generic_proxy_consumer =
     supplier_admin->obtain_notification_push_consumer(
       CosNotifyChannelAdmin::STRUCTURED_EVENT, proxy_id );
CosNotifyChannelAdmin::StructuredProxyPushConsumer_var
   proxy_push_consumer =
     CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(
       generic_proxy_consumer );
// Connect to the proxy so that we can post events.
proxy_push_consumer->connect_structured_push_supplier(
     CosNotifyComm::StructuredPushSupplier::_nil() );
...
// create an event
  CosNotification::StructuredEvent notification;
// set the domain to "News"
notification.header.fixed_header.event_type.domain_name =
    CORBA::string_dup("News");
// set the type to the news category
notification.header.fixed_header.event_type.type_name =
    CORBA::string_dup("Sports");
// add one field, which will contain the story, to the
// event's filterable data. set the field's name to
// "Story" and value to a string containing the story
notification.filterable_data.length(1);
notification.filterable_data[0].name =
           CORBA::string_dup("Story");
notification.filterable_data[0].value <<= "John Smith wins again";
// post the event
// Subscribers who subscribed to events whose domain is
// "News" and whose type matches the news category will
// receive this event
proxy_push_consumer->push_structured_event(notification);
...
```

```
// Disconnect.
proxy_push_consumer->disconnect_structured_push_consumer();
```

Step 2: Writing an Application to Subscribe to Events

The following types of CORBA applications can subscribe to events:

- C++ joint client/servers and servers.
- Foreign ORB clients that support callbacks.

To subscribe to events, an application must, at a minimum, support the following functions:

- Implement a CosNotifyComm OMG IDL interface that supports the `push_structured_event` operation.
- Get the event channel factory object reference and use it to get the event channel.
- Define and create a subscription that includes the callback object reference.
- Create a callback object that implements the `CosNotifyComm::StructuredPushConsumer` interface.

Implementing the CosNotifyComm::StructuredPushConsumer Interface

In order for the callback servant object to receive events, it must implement the CosNotifyComm::StructuredPushConsumer interface that supports the `push_structured_event` operation. When an event occurs that has a matching subscription, the Notification Service invokes this operation on the servant callback object in the subscriber application to deliver the event to the subscriber application.

The CosNotifyComm::StructuredPushConsumer interface also defines the operations `offer_change` and `disconnect_structured_push_consumer`. The Notification Service never invokes these operations, so you should implement stubbed out versions that throw `CORBA::NO_IMPLEMENT`.

Listing 4-3 and Listing 4-4 show how this interface is implemented in C++.

Listing 4-3 Sample CosNotifyComm::StructuredPushConsumer Interface Implementation (NewsConsumer_i.h)

```
#ifndef _news_consumer_i_h
#define _news_consumer_i_h
#include "CosNotifyComm_s.h"
// For the servant class to receive news events,
// it must implement the CosNotifyComm::StructuredPushConsumer
// idl interface
class NewsConsumer_i : public POA_CosNotifyComm::StructuredPushConsumer
{
  public:
    // this method will be called when a news event occurs
    virtual void push_structured_event(
      const CosNotification::StructuredEvent& notification
    );
    // OMG's CosNotifyComm::StructuredPushConsumer idl
    // interface defines the methods "offer_change" and
    // "disconnect_structured_push_consumer". Since the
    // Notification Service never invokes these methods, just
    // have them throw a CORBA::NO_IMPLEMENT exception

    virtual void offer_change(
      const CosNotification::EventTypeSeq& added,
      const CosNotification::EventTypeSeq& removed )
    {
      throw CORBA::NO_IMPLEMENT();
    }
    virtual void disconnect_structured_push_consumer()
    {
      throw CORBA::NO_IMPLEMENT();
    }
};
#endif
```

Listing 4-4 Sample CosNotifyComm::StructuredPushConsumer Interface Implementation (NewsConsumer_i.cpp)

```
#include "NewsConsumer_i.h"
#include <iostream.h>
//-------------------------------------------------------------
// Subscriber.cpp creates a simple events subscription to "News"
// events and has the events delivered to a NewsConsumer_i
// object. When a news event occurs (this happens when a user
// runs the Reporter application and reports a news story), this
// method will be invoked:
void NewsConsumer_i::push_structured_event(
  const CosNotification::StructuredEvent& notification )
  {
// extract the story from the first field in the event's
// filterable data
char* story;
notification.filterable_data[0].value >>= story;
// for coding simplicity, assume "story" is not "null"
// print out the event
cout
    << "-----------------------------------------------------"
    << endl
    << "Category : "
    << notification.header.fixed_header.
              v            event_type.type_name.in()
    << endl
    << "Story    : "
    << story
    << endl;
...
}
```

Getting the Event Channel, ConsumerAdmin Object, and Filter Factory Object

Before an application can create a subscription, it must get the event channel and the ConsumerAdmin and Filter Factory objects. Listing 4-5 shows how this is implemented in C++.

To get the event channel factory object reference, the `resolve_initial_references` method is invoked on the Bootstrap object using the `"NotificationService"` environmental object. The object reference is used to get the channel factory, which is, in turn, used to get the event channel. Finally, the event channel is used to get the ConsumerAdmin object and the FilterFactory object.

Listing 4-5 Getting the Event Channel and ConsumerAdmin and Filter Factory Objects (Subscriber.cpp)

```
// Get the CosNotification channel factory object reference.
CORBA::Object_var
  channel_factory_oref =
     bootstrap.resolve_initial_references(
          "NotificationService" );
   channel_factory =
      CosNotifyChannelAdmin::EventChannelFactory::_narrow(
          channel_factory_oref.in() );
// Use the channel factory to get the default channel.
CosNotifyChannelAdmin::EventChannel_var channel =
   channel_factory->get_event_channel(
      Tobj_Notification::DEFAULT_CHANNEL );
// Use the channel to get the consumer admin and the filter factory.
CosNotifyChannelAdmin::ConsumerAdmin_var consumer_admin =
   channel->default_consumer_admin();
CosNotifyFilter::FilterFactory_var filter_factory =
   channel->default_filter_factory();
```

Creating a Callback Object

To receive events, the application must also be a server; that is, the application must implement a callback object that can be invoked (called back) when an event occurs that matches the subscriber's subscription.

Creating a callback object includes the following steps:

Note: The following steps apply to a joint client/server. Oracle Tuxedo CORBA servers can also subscribe to events.

1. Creating a callback wrapper object. This can be implemented using either the BEAWrapper Callbacks object or the CORBA Portable Object Adaptor (POA).
2. Creating the servant.
3. Creating an object reference to the callback servant.

For a complete description of the BEAWrapper Callbacks object and its methods, see the Joint Client/Servers chapter in the *CORBA Programming Reference*.

Note: Using the BEAWrapper Callback object to create a callback object is discussed below. For a discussion of how to implement a callback object using the POA, see *Using CORBA Server-to-Server Communication.*

Listing 4-6 shows how to use the BEAWrapper Callbacks object to create a callback object in C++. In the code examples, the `NewsConsumber_i servant` is created and the `start_transient` method is used to create a transient object reference.

Listing 4-6 Sample Code for Creating a Callback Object with Transient Object Reference (Introductory Application Subscriber.cpp)

```
// Create a callback wrapper object since this client needs to
// support callbacks
BEAWrapper::Callbacks wrapper(orb.in());
NewsConsumer_i* news_consumer_impl = new NewsConsumer_i;
// Create a transient object reference to this servant.
CORBA::Object_var news_consumer_oref =
       wrapper.start_transient(
         news_consumer_impl,
         CosNotifyComm::_tc_StructuredPushConsumer->id()
          );
```

```
CosNotifyComm::StructuredPushConsumer_var
    news_consumer =
       CosNotifyComm::StructuredPushConsumer::_narrow(
           news_consumer_oref.in() );
```

Creating a Subscription

In order for the subscriber to receive events, it must subscribe to the Notification Service. You can create a transient subscription or a persistent subscription.

To create a subscription, the following steps must be performed:

1. Create a notification proxy push supplier and use it to create a StructuredProxySupplier object.
2. Set the subscription's Quality of Service (QoS). You can set the QoS to transient or persistent.
3. Create a filter object and assign the `domain_name`, `type_name`, and `data_filter` (optional) to it.
4. Add the filter to the proxy.
5. Connect to the proxy passing in the subscription's callback object reference.

Listing 4-7 from the Introductory sample application, shows how to create a transient subscription in C++.

Listing 4-7 Creating a Transient Subscription

```
// Create a new subscription (at this point, it is not complete).
CosNotifyChannelAdmin::ProxyID subscription_id;
CosNotifyChannelAdmin::ProxySupplier_var generic_subscription =
      consumer_admin->obtain_notification_push_supplier(
          CosNotifyChannelAdmin::STRUCTURED_EVENT,
          subscription_id );
CosNotifyChannelAdmin::StructuredProxyPushSupplier_var
     subscription =
       CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(
            generic_subscription );
```

```
    s_subscription = subscription.in();
// Set the quality of service. This sets the subscription name
// and subscription type (=TRANSIENT).
CosNotification::QoSProperties qos;
qos.length(2);
qos[0].name =
   CORBA::string_dup(Tobj_Notification::SUBSCRIPTION_NAME);
qos[0].value <<= subscription_name;
qos[1].name =
   CORBA::string_dup(Tobj_Notification::SUBSCRIPTION_TYPE);
qos[1].value <<=
   Tobj_Notification::TRANSIENT_SUBSCRIPTION;
subscription->set_qos(qos);
// Create a filter (used to specify domain, type and data filter).
CosNotifyFilter::Filter_var filter =
         filter_factory->create_filter(
          Tobj_Notification::CONSTRAINT_GRAMMAR );
s_filter = filter.in();
  // Set the filtering parameters.
  // (domain = "News", type = "Sports", and no data filter)
  CosNotifyFilter::ConstraintExpSeq constraints;
  constraints.length(1);
  constraints[0].event_types.length(1);
  constraints[0].event_types[0].domain_name =
        CORBA::string_dup("News");
  constraints[0].event_types[0].type_name =
        CORBA::string_dup("Sports");
  constraints[0].constraint_expr =
        CORBA::string_dup(""); // No data filter.
  CosNotifyFilter::ConstraintInfoSeq_var
      add_constraints_results = // ignore this returned value
          filter->add_constraints(constraints);
  // Add the filter to the subscription.
  CosNotifyFilter::FilterID filter_id =
     subscription->add_filter(filter.in());
  // Now that we have set the subscription name, type and filtering
  // parameters, complete the subscription by passing in the
  // reference of the callback object to deliver the events to.
```

```
subscription->connect_structured_push_consumer(
     news_consumer.in() );
```

Step 3: Compiling and Running Notification Service Applications

The final step in the development of a Notification Service application is to compile, build, and run the application. To do this, you need to perform the following steps.

1. Generate the required client stub and skeleton files to define interfaces between the Notification Service and event poster and subscriber applications. Event poster applications can be clients, joint client/servers, or servers. Event subscriber applications can be joint client/servers or servers.
2. Compile the application code and link against the skeleton and client stub files.
3. Build the application.
4. Run the application.

Generating the Client Stub and Skeleton Files

To generate the client stub and skeleton files, you must execute the `idl` command for each of the Notification IDL files that your application uses. Table 4-2 shows the `idl` commands used for each type of subscriber.

Table 4-2 idl Command Requirements

Language	Oracle Tuxedo CORBA Joint Client/Server	Oracle Tuxedo CORBA Server
C++	`idl -P`	`idl`

The following is an example of an `idl` command:

```
>idl -IC:\tuxdir\include C:\tuxdir\include\CosEventComm.idl
```

Table 4-3 lists the IDL files required by each type of Notification Service application.

Table 4-3 IDL Files Required by Notification Service Applications

Application Type	Required OMG IDL Files

Event poster (can be a client, a joint client/server, or a server)	`CosEventChannelAdmin.idl`
	`CosEventComm.idl`
	`CosNotification.idl`
	`CosNotifyChannelAdmin`
	`CosNotifyComm.idl`
	`CosNotifyFilter`
	`Tobj_Events.idl`
	`Tobj_Notification.idl`

Subscriber (can be joint client/server or a server)	`CosEventChannelAdmin.idl`
	`CosEventComm.idl`
	`CosNotification.idl`
	`CosNotifyChannelAdmin`
	`CosNotifyComm.idl`
	`CosNotifyFilter`
	`Tobj_Events.idl`
	`Tobj_Notification.idl`

Compiling and Linking the Application Code

The compiling and linking procedure differs depending on the type of Notification Service application you are building. Table 4-4 provides an overview of the commands and files used to compile each type of application.

Table 4-4 Application Build Requirements

Application Type	Client	Joint Client/Server	Server
C++ Events Poster	Use the `buildobjclient` command to compile the application files and the IDL stubs.	Use the `buildobjclient` command with the `-P` option to compile the application files and the IDL stubs.	Use the `buildobjserver` command to compile the application files and the IDL client stubs.
C++ Events Subscriber	Not applicable.	Use the `buildobjclient` command with the `-P` option to compile the application files, the IDL stubs, and the IDL skeletons.	Use the `buildobjserver` command to compile the application files, the IDL stubs, and the IDL skeletons.

Listing 4-8 shows the commands used for a C++ Reporter application (`Reporter.cpp`) on a Microsoft Windows system. To form a C++ executable, the `idl` command is run on the required IDL file and the `buildobjclient` command compiles the C++ client application file and the IDL stubs.

Listing 4-8 C++ Reporter Application Build and Run Commands

```
# Run the idl command.
idl -IC:\tuxdir\include C:\tuxdir\include\CosEventComm.idl \
C:\tuxdir\include\CosEventChannelAdmin \
C:\tuxdir\include\CosNotification.idl \
C:\tuxdir\include\CosNotifyComm.idl \
C:\tuxdir\include\CosNotifyFilter.idl \
C:\tuxdir\include\Tobj_Notification.idl
# Run the buildobjclient command.
buildobjclient -v -o is_reporter.exe -f "\
    -DWIN32                             \
    Reporter.cpp                        \
    CosEventComm_c.cpp                  \
    CosEventChannelAdmin_c.cpp          \
    CosNotification_c.cpp               \
```

```
    CosNotifyComm_c.cpp                  \
    CosNotifyFilter_c.cpp                \
    CosNotifyChannelAdmin_c.cpp          \
    Tobj_Events_c.cpp                    \
    Tobj_Notification_c.cpp "
# Run the application.
is_reporter
```

Listing 4-9 and Listing 4-10 show the commands used for a C++ Subscriber application (`Subscriber.cpp`) on Microsoft Windows and UNIX, respectively. To form a C++ executable, the `buildobjclient` command, with the `-P` option, compiles the joint client/server application files (`Subscriber.cpp` and `NewsConsumer_i.cpp`), the IDL stubs, the IDL skeleton (for `CosNotifyComm_s.cpp`).

Listing 4-9 C++ Subscriber Application Build and Run Commands (Microsoft Windows)

```
# Run the idl command.
idl -P -IC:\tuxdir\include C:\tuxdir\include\CosEventComm.idl \
C:\tuxdir\include\CosEventChannelAdmin \
C:\tuxdir\include\CosNotification.idl \
C:\tuxdir\include\CosNotifyComm.idl \
C:\tuxdir\include\CosNotifyFilter.idl \
C:\tuxdir\include\CosNotifyChannelAdmin \
\C:\tuxdir\include\Tobj_Events.idl \
\C:\tuxdir\include\Tobj_Notification
# Run the buildobjclient command.
buildobjclient -v -P -o is_subscriber.exe  -f " \
   -DWIN32                                    \
    Subscriber.cpp                             \
    NewsConsumer_i.cpp                         \
    CosEventComm_c.cpp                         \
    CosEventChannelAdmin_c.cpp                 \
    CosNotification_c.cpp                      \
    CosNotifyComm_c.cpp                        \
    CosNotifyComm_s.cpp                        \
    CosNotifyFilter_c.cpp                      \
```

```
    CosNotifyChannelAdmin_c.cpp                  \
    Tobj_Events_c.cpp                            \
    Tobj_Notification_c.cpp                      \
    C:\tuxdir\lib\libbeawrapper.lib              \
    "
# Run the application.
is_subscriber
```

Listing 4-10 C++ Subscriber Application Build and Run Commands (UNIX)

```
# Run the idl command.
idl -P -I/usr/local/tuxdir/include
/usr/local/tuxdir/include/CosEventChannelAdmin \
/usr/local/tuxdir/include/CosEventComm.idl \
/usr/local/tuxdir/include/CosNotification.idl \
/usr/local/tuxdir/include/CosNotifyComm.idl \
/usr/local/tuxdir/include/CosNotifyFilter.idl \
/usr/local/tuxdir/include/CosNotifyChannelAdmin \
/usr/local/tuxdir/include/Tobj_Events.idl \
/usr/local/tuxdir/include/Tobj_SimpleEvents.idl
# Run the buildobjclient command.
buildobjclient -v -P -o subscriber -f "     \
  Subscriber.cpp                            \
  NewsConsumer_i.cpp                        \
  CosEventComm_c.cpp                        \
  CosEventChannelAdmin_c.cpp                \
  CosNotification_c.cpp                     \
  CosNotifyComm_c.cpp                       \
  CosNotifyComm_s.cpp                       \
  CosNotifyFilter_c.cpp                     \
  CosNotifyChannelAdmin_c.cpp               \
  Tobj_Events_c.cpp                         \
  Tobj_SimpleEvents_c.cpp                   \
  -lbeawrapper                              \
"
```

```
# Run the application.
is_subscriber
```

CHAPTER 5

Building the Introductory Sample Application

This topic includes the following sections:

- Overview
- Building and Running the Introductory Sample Application

Note: The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Overview

The Introductory sample applications simulate a newsroom environment in which a news reporter posts a story and a news subscriber consumes the story.

One implementation of the Introductory sample application is provided: the C++ programming language that uses the Oracle Simple Events application programming interface (API). The Introductory sample application consists of the Reporter and Subscriber applications and the Notification Service. The Reporter application implements a client application that prompts the user to enter news articles, and then posts the news articles as events to the Oracle Tuxedo CORBA Notification Service. The Subscriber application implements a joint client/server

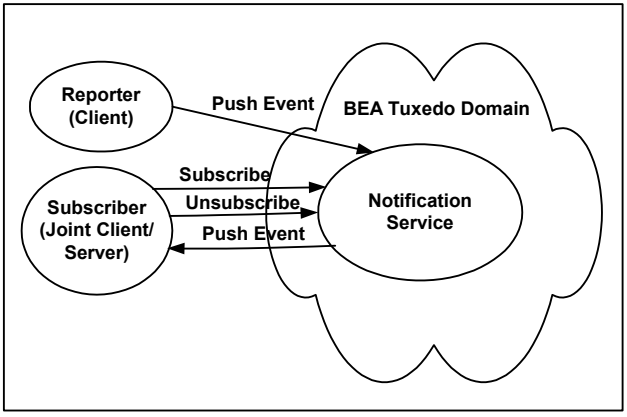
application that acts as client when it subscribes and unsubscribes for events, and acts as a server when it receives events. To receive events, the subscriber implements a callback object which is invoked by the Notification Service when an event needs to be delivered.

The Introductory sample application shows the simplest usage of the Notification Service. It demonstrates how to use the Oracle Simple Events API, the CosNotification API, transient subscriptions, and transient object references. It does not demonstrate the use of persistent subscriptions or data filtering. For a sample application that uses persistent subscriptions and data filtering, see Chapter 6, "Building the Advanced Sample Application."

This Introductory sample application provides two executables (see Figure 5-1):

- A Reporter application that posts events to the Notification Service. It is a client without callback capability.
- A Subscriber application that subscribes to the Notification Service and receives events. The subscriber is a joint client/server that acts as a client when it subscribes to events and acts as a server when it receives events.

Figure 5-1 Introductory Sample Application Components



The event poster, the Reporter application, uses the structured event `domain_name`, `type_name`, and `filterable_data` fields to construct the event. The domain name defines the industry. In this application, domain_name is set to "News". The `type_name` defines the kind of event in the industry and it is set to the category of news story (for example, "Sports"). The application user specifies this value. In the `filterable_data` fields, a field named "Story" is added, which contains the text of the news story being posted. This text is also specified by the application user.

The Subscriber application uses the structured event `domain_name` and `type_name` fields to create a subscription to the Notification Service. The subscription defines the `domain_name` as a fixed string with the content of "News". At run time, the Subscriber application queries the user for the "News Category" and uses the input to define the `type_name` field in the subscription. Obviously, the users of both applications, the reporter and the subscriber, must collaborate on the "News Category" string for the subscription to match an event, otherwise, no events will be delivered to the subscriber. The subscription does not do any checking of the `filterable_data` field, but rather assumes that the body of the story will be a string, and that the story will be in the first Named/Value pair in the `filterable_data` field of a structured event.

To post events, the Reporter application uses the `push_structured_event` method to push news events to the Notification Service. For each event, the Reporter application queries the user for a "News category" (for example, "Sports") and a story (a multiple-line text string).

To subscribe to news events, the Subscriber application invokes the Notification Service to subscribe to news events. For each subscription, the Subscriber application queries the user for a "News category" (for example, "Sports"). The Subscriber application also implements a callback object (via the NewsConsumer_i servant class) which is used to receive and process news events. When the Subscriber subscribes, it gives the Notification Service a reference to this callback object. When a matching event occurs; that is, when the Reporter posts an event with a "News category" that matches the news category of the subscription, the Notification Service invokes the `push_structured_event` method on the callback object to deliver the event to the callback object in the subscriber. This method prints out the event, invokes the `unsubscribe` method on the Notification Service to cancel the subscription, and shuts down the Subscriber. For simplicity, the `push_structured_event` method assumes that the `domain_name`, `type_name`, `length`, and `name` field match and the story is in the `value` field.

Note: The "News category" is just a string that the Reporter user and the Subscriber user agree on. There are no fixed categories in this sample. Therefore, both the Reporter user and the Subscriber user must type the same string when prompted for a category (including case and white space).

To run this sample, you must start at least one Reporter application and at least one Subscriber application; however, you may run multiple Reporters and Subscribers. Events posted by any Reporter will be delivered to all matching Subscribers (based on "News category").

Also, be sure to start any subscribers before posting events; otherwise, the events will be lost.

Building and Running the Introductory Sample Application

To build and run the Introductory sample application, you must perform these steps:

1. Verify that the "TUXDIR" environment variable are set to the correct directory path.
2. Unset "JAVA_HOME"
3. Copy the files for the Introductory sample application into a work directory.
4. Change the protection attributes on the files to grant write and execute access.
5. For UNIX, ensure the make file is in your path. For Microsoft Windows, ensure the nmake file is in your path
6. Set the application environment variables.
7. Build the sample.
8. Boot the system.
9. Run the Subscriber and Reporter applications.
10. Shut down the system.
11. Restore the directory to its original state.

These steps are described in detail in the following sections.

Verifying the Settings of the Environment Variables

Before you build and run the Introductory sample application, you need to ensure that the TUXDIR environment variable is set on your system. In most cases, this environment variable is set as part of the installation procedure. However, you need to check the environment variables to ensure they reflect correct information.

Table 5-1 lists the environment variables required to run the Introductory sample application.

Table 5-1 Required Environment Variables for the Introductory Sample Application

Environment Variable	Description
TUXDIR	The directory path where you installed the Oracle Tuxedo software. For example: **Windows** TUXDIR=c:\tuxdir **UNIX** TUXDIR=/usr/local/tuxdir

To verify that the information for the environment variables defined during installation is correct, perform the following steps:

Windows

1. From the Start menu, select Settings.
2. From the Settings menu, select the Control Panel.

   The Control Panel appears.
3. Click the System icon.

   The System Properties window appears.
4. Click the Environment tab.

   The Environment page appears.
5. Check the setting for TUXDIR

UNIX

```
ksh prompt>printenv TUXDIR
```

To change the settings, perform the following steps:

Windows

1. On the Environment page in the System Properties window, click the environment variable you want to change.
2. Enter the correct information for the environment variable in the Value field.
3. Click OK to save the changes.

UNIX

```
ksh prompt>export TUXDIR=directorypath
```

Or

```
csh> setenv TUXDIR=directorypath
```

Copying the Files for the Introductory Sample Application into a Work Directory

You need to copy the files for the Introductory sample application and files in the common directory into a work directory on your local machine.

Note: The application directory and the common directory must be copied to the same parent directory.

The files are located in the following directories:

Windows

For the C++ Introductory sample:

```
drive:\tuxdir\samples\corba\notification\introductory_simple_cxx
drive:\tuxdir\samples\corba\notification\common
```

UNIX

For the C++ Introductory sample:

```
/usr/local/tuxdir/samples/corba/notification/
introductory_simple_cxx
/usr/local/tuxdir/samples/corba/notification/common
```

You use the files listed in Table 5-2 and Table 5-3 to build and run the C++ Introductory sample application, which is implemented using the Oracle Simple Events API.

Table 5-2 Files Located in the introductory_sample_c++ Directory

File	Description
`Readme.txt`	Describes the Introductory sample application and provides instructions for setting up the environment and building and running the application.
`setenv.cmd`	Sets the environment for Microsoft Windows systems.
`setenv.ksh`	Sets the environment for UNIX systems.

Table 5-2 Files Located in the introductory_sample_c++ Directory (Continued)

File	Description
`makefile.nt`	Makefile for Microsoft Windows systems.
`makefile.mk`	Makefile for UNIX systems.
`makefile.inc`	Common makefile used by the `makefile.nt` and the `makefile.mk` files.
`Reporter.cpp`	Code for the reporter.
`Subscriber.cpp`	Code for the subscriber.
`NewsConsumer_i.h` and `NewsConsumer.cpp`	The callback servant class that subscribers use to receive news events. (For the Subscriber application.)

Table 5-3 lists other files that the Introductory sample application uses.

Table 5-3 Other Files the Introductory Sample Application Uses

File	Description
The following files are located in the common directory.	
`common.nt`	Makefile symbols for Microsoft Windows systems.
`common.mk`	Makefile symbols for UNIX systems.
`introductory.inc`	Makefile for administrative targets.
`ex.h`	Utilities to print exceptions. (For C++ only.)
`client_ex.h`	Client utilities to handle exceptions. (For C++ only.)
The following files are located in the \tuxdir\include directory.	
`CosEventComm.idl`	The OMG IDL code that declares the CosEventComm module.
`CosNotification.idl`	The OMG IDL code that declares the CosNotification module.
`CosNotifyComm.idl`	The OMG IDL code that declares the CosNotifyComm module.

Table 5-3 Other Files the Introductory Sample Application Uses (Continued)

File	Description
`Tobj_Events.idl`	The OMG IDL code that declares the Tobj_Events module.
`Tobj_SimpleEvents.idl`	The OMG IDL code that declares the Tobj_SimpleEvents module. **Note:** This file is needed only for the application that was developed using Oracle Simple Events API.
The following files are needed only for the application that was developed using CosNotification Service API.	
`CosEventChannelAdmin.idl`	The OMG IDL code that declares the CosEventChannelAdmin module.
`CosNotifyFilter.idl`	The OMG IDL code that declares the CosNotifyFilter module.
`CosNotifyChannelAdmin.idl`	The OMG IDL code that declares the CosNotifyChannelAdmin module.
`Tobj_Notification.idl`	The OMG IDL code that declares the Tobj_Notification module.

Changing the Protection Attribute on the Files for the Introductory Sample Application

During the installation of the Oracle Tuxedo CORBA software, the sample application files are marked read-only. Before you can edit or build the files in the Introductory sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

Windows

1. In a DOS window, change (`cd`) to your work directory.
2. `prompt>attrib -r` *`drive:\workdirectory`*`\*.*`

UNIX

1. Change (cd) to your work directory.

```
2. prompt>/bin/ksh
```

```
3. ksh prompt>chmod u+w /workdirectory/*.*
```

On UNIX systems, you also need to change the permission of `setenv.ksh` to give execute permission to the file, as follows:

```
ksh prompt>chmod +x setenv.ksh
```

Setting Up the Environment

To set up the environment, enter the following command:

Windows

```
prompt>.\setenv.cmd
```

UNIX

```
ksh prompt>. ./setenv.ksh
```

Building the Introductory Sample Application

You use the `make` command to run makefiles, which are provided for Microsoft Windows and UNIX, to build the sample application. For UNIX, use `make`. For Microsoft Windows, use `nmake`.

Makefile Summary

The makefile automates the following steps:

1. Checks that the set environment command (`setenv.cmd`) has been run. If the environment variables have not been set, the makefile prints an error message to the screen and exits.
2. Includes the `common.nt` (for Microsoft Windows) or `common.mk` (for UNIX) command file. This file defines the makefile symbols used by the samples. These symbols allow the UNIX and Microsoft Windows makefiles to delegate the build rules to platform-independent makefiles.
3. Includes the `makefile.inc` command file. This file builds the `is_reporter` and `is_subscriber` executables, and cleans up the directory of unneeded files and directories.
4. Includes the `introductory.inc` command file. This file creates the `UBBCONFIG` file and executes the `tmloadcf -y ubb` command to create the `TUXCONFIG` file. This is a

platform-independent makefile fragment that defines the administrative build rules common to the Introductory sample application.

Executing the Makefile

Before executing the `makefile`, you need to check the following:

- Ensure that you have the appropriate administrative privileges to build and run applications.
- On Microsoft Windows, verify that `nmake` is in the path of your machine.
- On UNIX, verify that `make` is in the path of your machine.

To build the Introductory sample application, enter the `make` command as follows:

Windows

```
nmake -f makefile.nt
```

UNIX

```
make -f makefile.mk
```

Starting the Introductory Sample Application

To start the Introductory sample application, enter the following commands:

1. To boot the Oracle Tuxedo system:

   ```
   prompt>tmboot -y
   ```

   This command starts the following server processes:

   - `TMSUSREVT`

     An Oracle Tuxedo system-provided, EventBroker server that is used by the Notification Service.

   - `TMNTS`

     An Oracle Tuxedo Notification Service server that processes requests for subscriptions and event postings.

   - `TMNTSFWD_T`

     An Oracle Tuxedo Notification Service server that forwards events to subscribers that have transient subscriptions.

   - `ISL`

The IIOP Listener/Handler process.

2. To start the Subscriber application:

   For C++: `prompt>is_subscriber`

   To start another Subscriber, open another window, change (`cd`) to your work directory, set the environment variables (by running `setenv.cmd` or `setenv.ksh`), and enter the start command that is appropriate for your platform.

3. To start the Reporter application, open another window and enter the following:

   For C++: `prompt>is_reporter`

   To start another Reporter, open another window, change (`cd`) to your work directory, set the environment variables (by running `setenv.cmd` or `setenv.ksh`), and enter the start command that is appropriate for your platform.

Using the Introductory Sample Application

To use the Introductory sample application, you must use the Subscriber application to subscribe to an event and the Reporter application to post an event. Be sure to subscribe before you post each event; otherwise, events will be lost.

Note: The Subscriber application shuts down after it receives one event.

Using the Subscriber Application to Subscribe to Events

Perform these steps:

1. When you start the Subscriber application (`prompt>is_subscriber`), the following prompts are displayed:

   ```
   Name?                  (Enter a name (without spaces).)
   Category (or all)?     (Enter the category of news you want or "all".)
   ```

   You may type in any string for the news category; that is, there is no fixed list of news categories. However, when you use the Reporter application to post an event, make sure to specify the same string for the news category.

2. The Subscriber application creates a subscription then prints “Ready” when it is ready to receive events. After the Subscriber receives one event, it shuts down.

   Note: You should always use the Subscriber application to subscribe to events before you use the Reporter application to post events; otherwise, events will be lost.

Using the Reporter Application to Post Events

Perform these steps:

1. When you start the Reporter application (`prompt> is_reporter`), the following prompts are displayed:

   ```
   (r) Report news
   (e) Exit

   Option?
   ```

2. Enter `r` to report news. The following prompt is displayed:

   ```
   Category?
   ```

3. Enter the news category. It must match exactly the category you typed on the Subscriber application (including white space and case).

   After you enter the news category, the following prompt is displayed:

   ```
   Enter story (terminate with '.')
   ```

4. Enter your story. It can span multiple lines. Finish the story by typing a period only (`"."`) on a line, followed by a carriage return.

   Subscribers whose category matches the category of this story will receive, and print out the story. When a subscriber receives a story, the subscriber automatically shuts down.

5. To send and receive more news stories, start another subscriber, then report another story. When you are done reporting news, choose the Exit `(e)` option.

   Note: The Subscriber application shuts down after it receives one event. Therefore, always use the Subscriber application to subscribe to events before you use the Reporter application to post an event; otherwise, events will be lost.

Shutting Down the System and Cleaning Up the Directory

Perform the following steps:

Note: Make sure the Reporter and Subscriber processes have stopped.

1. To shut down the system, in any window, type:

   ```
   prompt>tmshutdown -y
   ```

2. To restore the directory to its original state, in any window, type:

   Windows

```
prompt>nmake -f makefile.nt clean
```

UNIX

```
prompt>make -f makefile.mk clean
```

CHAPTER 6

Building the Advanced Sample Application

This topic includes the following sections:

- Overview
- Building and Running the Advanced Sample Application

Note: The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Overview

The Advanced sample application simulates a newsroom environment in which a news reporter posts a story, a wire service posts the story as an event to the Notification Service, and a news subscriber consumes the story.

One implementation of the Advanced sample application is provided: the C++ that uses the CosNotification Service API.

The Advanced sample application consists of the reporter, subscriber, and wire service applications that use the Oracle Tuxedo CORBA Notification Service. The reporter application implements a client application. This application prompts the user to enter news articles and calls

the WireService server using application specific IDL. The WireService server, in turn, posts the events. The subscriber implements a joint client/server application. This application acts as client when it subscribes and unsubscribes for events, and acts as a server when it receives events. To receive events, the Subscriber implements callback objects which are invoked by the Notification Service when an event needs to be delivered.

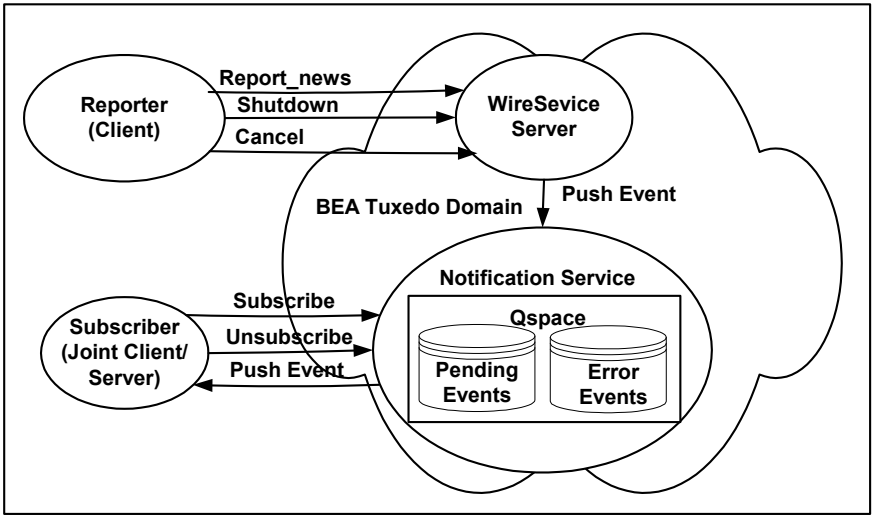
Note: On UNIX systems, you cannot immediately restart the subscriber because the port takes some time (the actual time depends on the platform) to become available again. If you restart too soon, you will get a `CORBA::OBJ_ADAPTER` exception. If this occurs, just wait and try again. On Solaris systems, the port can take up to 10 minutes to become available. To see if the port is still in use, use this command: "`Restart -a | grep <the port number>`".

This Advanced sample application demonstrates how to use the Oracle Simple Events API, the CosNotification Service API, transient and persistent subscriptions, and data filtering.

This Advanced sample provides three executables (see Figure 6-1):

- A WireService application that posts events. It is a Notification Service client and an Oracle Tuxedo CORBA server. It implements an OMG IDL interface, which the Reporter application uses.
- A Reporter application that reports news stories by invoking methods on the WireService. The WireService, in turn, converts the stories into events and posts them using the Notification Service. The reporter is a pure client.
- A Subscriber application that subscribes to the Notification Service and receives events. The subscriber is a joint client/server that acts as a client when it subscribes for events, and acts as a server when it receives events.

Figure 6-1 Advanced Sample Application Components



The event poster, the WireService application, uses the structured event `domain_name`, `type_name`, and `filterable_data` fields to construct three events: a news event, a subscriber shutdown event, and a subscriber cancel event.

- News event

  For this event, the domain name is a string and is preset by the application as “News”. The type name is a string and defined by the Reporter application user at run time. It is set to the category of news (for example, “Sports”). Filterable data contains a name/value pair whose name is “Story” and whose value is a string that contains the body of the news story being posted.

- Subscriber Shutdown event

  For this event, the domain name is a string and is preset by the application as “NewsAdmin”. The type name is a string and is preset by the application as “Shutdown”. The filterable data is not used.

- Subscriber Cancel event

  For this event, the domain name is a string and is preset by the application as “NewsAdmin”. The type name is a string and is preset by the application as “Cancel”. The filterable data is not used.

The Subscriber application uses the structured event `domain_name`, `type_name`, and `filterable_data` fields to construct two subscriptions: a news subscription that processes news stories; and a shutdown subscription that processes Cancel and Shutdown events. At run time, the Subscriber application establishes these two subscriptions with the Notification Service.

- News subscription

  The Subscriber application uses the structured event `domain_name`, `type_name`, and `filterable_data` fields to create a subscription to the Notification Service. The subscription defines the domain name as a fixed string with the content of "News". At run time, the Subscriber application queries the user for the "News Category" and "Keyword" and uses the inputs to define the type_name and data_filter fields in the subscription. Obviously, the users of both applications, the reporter and the subscriber, must collaborate on the "News Category" and "keyword" strings for the subscription to match an event, otherwise, no News events will be delivered to the subscriber. The subscription does not do any checking of the `filterable_data` field, but rather assumes that the body of the story will be a string, and that the story will be in the first Named/Value pair in the `filterable_data` field of a structured event.

- Shutdown subscription

  The Subscriber application uses the structured event domain_name and type_name, fields to create a subscription to the Notification Service. The subscription defines the domain_name as a fixed string with the content of "NewsAdmin", the type_name as a string of either "Shutdown" or "Cancel". The filterable_data field is an empty string.

The Reporter application is responsible for implementing the user interface for reporting news as well as for producing Shutdown and Cancel events. Rather than use the Notification Service directly to post events, it calls methods on the WireService server.

The WireService server uses the Notification Service to post three kinds of events:

- "News" events (used to deliver news to subscribers)
- "Shutdown" events (used to shut down subscribers temporarily)
- "Cancel" events (used to shut down subscribers permanently)

The Notification Service, in turn, delivers the events to the subscribers.

The subscriber uses the Notification Service to create a persistent subscription to news events. The subscriber implements a persistent callback object (via the `NewsConsumer_i` servant class), which is used to receive and process news events. When the subscriber subscribes, it gives the Notification Service a reference to this callback object. When a matching event occurs, the

Notification Service invokes a `push_structured_event` method on this callback object to push the event to the subscriber. This method prints out the event.

The subscriber also uses the Notification Service to create a transient subscription to Shutdown and Cancel events. The subscriber implements another callback object (via the `ShutdownConsumer_i` servant class), which is used to receive and process these events.

Whenever the subscriber runs, it prompts the user for a name. The first time this user runs the subscriber program, the subscriber creates a persistent subscription to News events. To do this, the subscriber prompts the user for which kind of news stories to subscribe to and which port number the subscriber should run on. The subscriber runs on this port, subscribes, then writes the subscription ID, the filter ID (if using the CosNotification API), and the port number to a file (the name of the file is `<`*`user_name`*`>.pstore`). The next time the subscriber runs, the subscriber prompts the user for a name, opens up the file `<`*`user_name`*`>.pstore` then reads the subscription ID, filter ID (if using the CosNotification API) and port number for this user from the file. This satisfies the requirement that the subscriber runs on the same port number each time because its news callback object's object reference is persistent.

The Subscriber creates a transient subscription to receive the Shutdown and Cancel events, therefore, the transient subscription is created and destroyed every time the subscriber is run and shut down. This subscription ID is not written out to the file `<`*`user_name`*`>.pstore`.

When the subscriber receives a Shutdown event, it destroys the shutdown/callback subscription but leaves the News subscription intact. If News events are posted after the subscriber is shut down and before it is restarted, then the notification service will either deliver the events when the subscriber is restarted, or will put the events on the error queue. (You can use the `ntsadmin` utility to either delete these events from the error queue or retry delivering them.)

Whether the event is redelivered or is put on the error queue depends on whether the subscriber restarts quickly enough. This depends on the retry parameters of the queue. See `advanced.inc` (in the notification samples' common directory) for the values of the queue retry parameters.

News events have two parts: a category (for example, headline) and a story (a multiple-line text string). The Subscriber application prompts the user to input a news category. Next the subscriber subscribes to news events whose category matches this string. The Reporter application prompts the user for a news category and a story. Next the reporter (by invoking a method on the wire service) posts a corresponding news event. The event will only be delivered to subscribers who subscribed to that category of news.

Note: The category is a string. The same string must be used by the Reporter user and the Subscriber user. There are no fixed categories in this sample. Therefore both users, the

Reporter user and the Subscriber user, must type the same string when prompted for a category (including case and white space).

This sample also uses data filtering. When a user first runs the Subscriber, the user will be prompted for a "keyword." Events whose category matches *and* whose story contains the keyword will be delivered to the subscriber. For example, if the user enters a keyword of "none," data filtering will not be used (thus the user will receive all events for the chosen news category). If the user enters a keyword "smith", it translates to `"Story %% '.*smith.*'"`. This keyword specifies that the subscription only accepts events that have a "Story" field that contains a string, and that the field starts with any number of characters, has a literal string "smith", and then ends with any number of characters.

To run this sample, you need to run at least one Reporter and at least one Subscriber; however, you may run multiple Reporters and multiple Subscribers. Events posted by any Reporter will be delivered to all matching Subscribers (based on the category).

Also, be sure to start any subscribers before posting events. Events posted before the subscribers are started will not be delivered.

Building and Running the Advanced Sample Application

To build and run the Introductory sample application, you must perform these steps:

1. Verify that the `"TUXDIR"` environment variable is set to the correct directory path.
2. Unset and `"JAVA_HOME"`
3. Copy the files for the Introductory sample application into a work directory.
4. Change the protection attributes on the files to grant write and execute access.
5. For UNIX, ensure the `make` file is in your path. For Microsoft Windows, ensure the `nmake` file is in your path
6. Set the application environment variables.
7. Build the sample.
8. Boot the system.
9. Run the Subscriber and Reporter applications.
10. Shut down the system.
11. Restore the directory to its original state.

These steps are described in detail in the following sections.

Verifying the Settings of the Environment Variables

Before you build and run the Advanced sample application, you need to ensure that the TUXDIR environment variable is set on your system. In most cases, this environment variable is set as part of the installation procedure. However, you need to check the environment variables to ensure they reflect the correct information.

Table 6-1 lists the environment variables required to run the Callback sample application.

Table 6-1 Required Environment Variables for the Callback Sample Application

Environment Variable	Description
TUXDIR	The directory path where you installed the Oracle Tuxedo software. For example: **Windows** TUXDIR=c:\tuxdir **UNIX** TUXDIR=/usr/local/tuxdir

To verify that the information for the environment variables defined during installation is correct, perform the following steps:

Windows

1. From the Start menu, select Settings.
2. From the Settings menu, select the Control Panel.

   The Control Panel appears.
3. Click the System icon.

   The System Properties window appears.
4. Click the Environment tab.

   The Environment page appears.
5. Check the setting for TUXDIR

UNIX

```
ksh prompt>printenv TUXDIR
```

To change the settings, perform the following steps:

Windows

1. On the Environment page in the System Properties window, click the environment variable you want to change.
2. Enter the correct information for the environment variable in the Value field.
3. Click OK to save the changes.

UNIX

```
ksh prompt>export TUXDIR=directorypath
```

Copying the Files for the Advanced Sample Application into a Work Directory

You need to copy the files for the Advanced sample application into a work directory on your local machine.

Note: The application directory and the common directory must be copied to the same parent directory.

The files for the Advanced sample application are located in the following directories:

Windows

For the C++ Advanced sample:

```
drive:\tuxdir\samples\corba\notification\advanced_cos_cxx
drive:\tuxdir\samples\corba\notification\common
```

UNIX

For the C++ Advanced sample:

```
/usr/local/tuxdir/samples/corba/notification/advanced_cos_cxx
/usr/local/tuxdir/samples/corba/notification/common
```

You use the files listed in Table 6-2 and Table 6-3 to build and run the C++ Advanced sample application, which is implemented using the CosNotification API.

You use the files listed in Table 6-2 and Table 6-3 to build and run the Advanced sample application.

Table 6-2 Files Located in the advanced_cos_c++ Notification Directory

File	Description
`Readme.txt`	Describes the Advanced sample application and provides instructions for setting up the environment and building and running the application.
`setenv.cmd`	Sets the environment for Microsoft Windows systems.
`setenv.ksh`	Sets the environment for UNIX systems.
`makefile.nt`	Makefile for Microsoft Windows systems.
`makefile.mk`	Makefile for UNIX systems.
`makefile.inc`	Common makefile used by the `makefile.nt` and the `makefile.mk` files.
`Reporter.cpp`	Code for the reporter.
`Subscriber.cpp`	Code for the subscriber.
`NewsConsumer_i.h` and `NewsConsumer.cpp`	Callback servant class that subscribers use to receive news events. (For the Subscriber application.)
`ShutdownConsumer_i.h` and `ShutdownConsumer.cpp`	Callback servant classes that subscribers use to receive Shutdown and Cancel events. (For the Subscriber application.)
`WireServiceServer.cpp`	Code for the WireService server.
`News.icf`	ICF file for the WireService interfaces.
`WireService_i.h` and `WireService.cpp`	Implements the WireService interfaces.

Table 6-3 lists other files that the Advanced sample application uses. With the exception of the IDL files, the files are located in the Notification common directory.

Table 6-3 Other Files That the Advanced Sample Uses

File	Description
The following files are located in the common directory.	
`News.idl`	IDL definitions for the WireService server.
`news_flds`	FML field definitions used to perform data filtering and news events.
`common.nt`	Makefile symbols for Microsoft Windows systems.
`common.mk`	Makefile symbols for UNIX systems.
`advanced.inc`	Makefile for administrative targets.
`ex.h`	Utilities to print exceptions (C++ only).
`client_ex.h`	Client utilities to handle exceptions (C++ only).
`server_ex.h`	Server utilities to handle exceptions.
The following files are located in the \tuxdir\include directory.	
`CosEventComm.idl`	The OMG IDL code that declares the CosEventComm module.
`CosNotification.idl`	The OMG IDL code that declares the CosNotification module.
`CosNotifyComm.idl`	The OMG IDL code that declares the CosNotifyComm module.
`Tobj_Events.idl`	The OMG IDL code that declares the Tobj_Events module.
`Tobj_SimpleEvents.idl`	The OMG IDL code that declares the Tobj_SimpleEvents module. **Note:** This file is needed only for the application that was developed using Oracle Simple Events API.
The following files are needed only for the application that was developed using CosNotification Service API.	

Table 6-3 Other Files That the Advanced Sample Uses (Continued)

File	Description
`CosEventChannelAdmin.idl`	The OMG IDL code that declares the CosEventChannelAdmin module.
`CosNotifyFilter.idl`	The OMG IDL code that declares the CosNotifyFilter module.
`CosNotifyChannelAdmin.idl`	The OMG IDL code that declares the CosNotifyChannelAdmin module.
`Tobj_Notification.idl`	The OMG IDL code that declares the Tobj_Notification module.

Changing the Protection Attribute on the Files for the Advanced Sample Application

During the installation of the Oracle Tuxedo software, the Advanced sample application files are marked read-only. Before you can edit or build the files in the Advanced sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

Windows

1. Change (`cd`) to your work directory
2. `prompt>attrib -r` *`drive:\workdirectory\*.*`*

UNIX

1. Change (`cd`) to your work directory
2. `prompt>/bin/ksh`
3. `ksh prompt>chmod u+w /`*`workdirectory`*`/*.*`

On the UNIX operating system platform, you also need to change the permission of `setenv.ksh` to give execute permission to the file, as follows:

```
ksh prompt>chmod +x setenv.ksh
```

Setting Up the Environment

To set up the environment, enter the following command:

Windows

```
prompt>.\setenv.cmd
```

UNIX

```
prompt>. ./setenv.ksh
```

Building the Advanced Sample Application

You use the `make` command to run `makefiles`, which are provided for Microsoft Windows and UNIX, to build the sample application. For Microsoft Windows, use `nmake`. For UNIX, use `make`.

Makefile Summary

The makefile automates the following steps:

1. Checks that the set environment command (`setenv.cmd`) has been run. If the environment variables have not been set, the makefile prints an error message to the screen and exits.
2. Includes the `common.nt` (for Microsoft Windows) or `common.mk` (for UNIX) command file. This file defines the makefile symbols used by the samples. These symbols allow the UNIX and Microsoft Windows makefiles to delegate the build rules to platform-independent makefiles.
3. Includes the `makefile.inc` command file. This file builds the `is_reporter`, `is_subscriber` and `AS_WIRESERVICE` executables, and cleans up the directory of unnecessary files and directories.
4. Includes the `advanced.inc` command file. This file executes `tmadmin` and `qadmin` commands to create the transaction log and the queues required by the persistent subscriptions. It also creates the `UBBCONFIG` file and executes the `tmloadcf -y ubb` command to create the `TUXCONFIG` file.

Executing the Makefile

Before executing the `makefile`, you need to check the following:

- Ensure that you have the appropriate administrative privileges to build and run applications.

- On Microsoft Windows, make sure `nmake` is in the path of your machine.
- On UNIX, make sure `make` is in the path of your machine.

To build the Advanced sample application, enter the `make` command as follows:

Windows

```
nmake -f makefile.nt
```

UNIX

```
make -f makefile.mk
```

Starting the Advanced Sample Application

To start the Advanced sample application, enter the following commands:

1. To boot the Oracle Tuxedo system:

   ```
   prompt>tmboot -y
   ```

   This command starts the following server processes:

   - `TMSUSREVT`

     An Oracle Tuxedo system-provided, EventBroker server that is used by the Notification Service.

   - `TMNTS`

     An Oracle Tuxedo CORBA Notification Service server that processes requests for subscriptions and event postings.

   - `TMNTSFWD_T`

     An Oracle Tuxedo CORBA Notification Service server that forwards events to subscribers that have transient subscriptions. This server is required for transient subscriptions.

   - `TMNTSFWD_P`

     An Oracle Tuxedo CORBA Notification Service server that forwards persistent events to subscribers that have persistent subscriptions. This server is required for persistent subscriptions.

   - `TMQUEUE`

     The message queue manager is an Oracle Tuxedo system-provided server that enqueues and dequeues messages on behalf of programs calling `tpenqueue`(3) and `tpdequeue`(3), respectively. This server is required for persistent subscriptions.

- TMQFORWARD

  The message forwarding server is an Oracle Tuxedo system-provided server that forwards messages that have been stored using tpenqueue(3c) for later processing. This server is required for persistent subscriptions.

- WIRE_SERVICE_SERVER

  A server, specifically built for the Advanced sample application, that receives events from the Reporter application and posts them to the Notification Service. This receive and server posts three types of events: News, Shutdown, and Cancel.

- ISL

  The IIOP Listener/Handler process.

2. To start the Subscriber application:

   For C++: `prompt>is_subscriber`

   To start another Subscriber, open another window, change (`cd`) to your work directory, set the environment variables (by running `setenv.cmd` or `setenv.ksh`), and enter the start command that is appropriate for your platform.

3. To start the Reporter application, open another window and enter the following:

   For C++: `prompt>is_reporter`

   To start another Reporter, open another window, change (`cd`) to your work directory, set the environment variables (by running `setenv.cmd` or `setenv.ksh`), and enter the start command that is appropriate for your platform.

Using the Advanced Sample Application

To use the Advanced sample application, you must use the Subscriber application to subscribe to an event and the Reporter application to post to an event. Be sure to subscribe before you post each event; otherwise, events will be lost.

Using the Subscriber Application to Subscribe to Events

Perform the following steps:

1. When you start the Subscriber application (`prompt>is_subscriber`) for the first time, the following prompts are displayed:

```
Name?                (Enter a name (without spaces).)
Port (e.g. 2463)     (Enter the port number that this subscriber should run on.)
Category (or all)    (Enter the category of news you want or "all.")
Keyword (or none)    (Enter a keyword that you want all delivered stories to
                      contain.)
```

Note: If the Subscriber application is shut down by a Shutdown event from the Reporter application (Shutdown events do not cancel persistent subscriptions), on subsequent startups of the Subscriber application, you will only be prompted for your name. The Subscriber application retrieves the remaining information from the `<user_name>.pstore` file. This guarantees that the same port number is used, which is required for persistent subscriptions.

If the Subscriber application is shut down by a Cancel event from the Reporter application (Cancel events cancel all subscriptions including persistent subscriptions), on subsequent startups of the Subscriber application, you will be prompted for your name, port number, category, and keyword.

2. You may type in any string for the news category, that is, there is no fixed list of news categories. However, when you use the Reporter application to post an event, make sure you specify the same string for the news category.

   Similarly, you may type in a string for a keyword. There is no fixed list of keywords either so when you run the reporter and enter the story, make sure that the story contains the same string; otherwise, the story will not be delivered to your subscription.

   The first time the Subscriber application is run for your username, category (or all), and keyword (optional), it creates a news subscription. On subsequent runs, the subscriber reuses this subscription. In all cases, the Subscriber application prints “Ready” when it is ready to receive events.

   The Subscriber application creates a subscription then prints “Ready” when it is ready to receive events.

   Note: You should always use the Subscriber application to subscribe to events before you use the Reporter application to post events; otherwise, events will be lost. This is because even though the Subscriber application creates a persistent subscription to News events, that subscription is not created until the Subscriber application is started.

   Note: You can start multiple subscribers by opening another window and repeating this procedure.

Using the Reporter Application to Post Events

Perform the following steps:

1. When you start the Reporter application (`prompt> is_reporter`), the following prompt is displayed:

   ```
   (r) Report news
   (s) Shutdown subscribers
   (c) Cancel Subscribers
   (e) Exit

   Option?
   ```

2. Enter `r` to report news. The following prompt is displayed:

   ```
   Category?
   ```

3. Enter the news category. It must match exactly the category you typed on the Subscriber application (including white space and case).

   After you enter the news category, the following prompt is displayed:

   ```
   Enter story (terminate with '.')
   ```

4. Enter your story. It can span multiple lines. Finish the story by typing a period only (".") on a line, followed by a carriage return. If you typed in a keyword when subscribing, make sure the story contains this string (including white space and case).

   Subscribers whose category and keyword (if specified) matches the category and a keyword in this story will receive and print out the story.

5. If you choose the "`s`" option, a Shutdown event will be posted and received by all the subscribers and the subscribers will shut down. While the subscribers are shut down, you may post another news story (by using the "`r`" option again). The Notification Service will place the news story on the pending queue but the News event subscription is persistent and, therefore, is still in effect. After you restart the subscribers, they will receive this second news story (unless a restart delay caused the event to be moved to the error queue). This is because the subscriber created a persistent subscription for news stories.

   Note: You can use the `ntsadmin retryerrevents` command to move events from the error queue back to the pending queue.

6. If you choose the "`c`" option, a Cancel event will be posted and received by all the subscribers. The subscribers will cancel their news subscriptions and shut down. If you try to restart the subscribers, then you will be prompted again for port, category, and keyword because you are creating a new subscription.

7. When you are finished reporting news, choose the Exit `(e)` option.

   Note: You can start multiple reporters by opening another window and repeating this procedure. Any news story reported by any reporter will be delivered to all matching subscribers. Make sure you have exited all reporters before shutting down the system.

Shutting Down the System and Cleaning Up the Directory

Make sure the Reporter and Subscriber processes have stopped and perform the following steps:

1. To shut down the system, in any window, type:

   ```
   prompt>tmshutdown -y
   ```

2. To restore the directory to its original state, in any window, type:

   Windows

   ```
   prompt>nmake -f makefile.nt clean
   ```

   UNIX

   ```
   prompt>make -f makefile.mk clean
   ```

CHAPTER 7

CORBA Notification Service Administration

This topic includes the following sections:

- Introduction
- Configuring the Notification Service. This section includes the following topics:
  - Configuring Data Filters
  - Setting the Host and Port
  - Creating a Transaction Log
  - Creating Event Queues
  - Creating the UBBCONFIG File and the TUXCONFIG File
- Managing the Notification Service
- Notification Service Administration Utility and Commands
- Notification Servers

Note: The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Introduction

The Oracle Tuxedo CORBA Notification Service is layered on the Oracle Tuxedo EventBroker and Queuing systems. This means that administering the CORBA Notification Service requires that you also administer these other Oracle Tuxedo systems. You use the Oracle Tuxedo utilities `tmadmin`, `qmadmin`, and `ntsadmin` to administer the Notification Service.

Notification Service administration is comprised of two related tasks: configuration and management. Although these areas are discussed separately, they are in fact, interrelated. Thus, to fully understand configuration, you must also understand management and vice versa.

Configuring the Notification Service

Before you can run event Notification Service applications, the following configuration requirements must be satisfied:

- If data filtering or Oracle Tuxedo ATMI interoperability is to be used, create Oracle Tuxedo ATMI FML field definition files that describe the fields on which to filter or to interoperate.
- If persistent subscriptions are to be used:
  - If using a a joint client/server, set the host and port number for the callback object references.
  - Create a transaction log.
  - Create queues to hold events.
- Create a system configuration file (`UBBCONFIG`) and a `TUXCONFIG` file.

Configuring Data Filters

If data filtering or Oracle Tuxedo ATMI interoperability is used in subscriber applications, you must perform the following steps to use data filtering in subscriptions:

1. Create the Oracle Tuxedo ATMI FML field table definition file that describes the fields on which to filter (see Listing 7-2).
2. In the `UBBCONFIG` file, specify where the FML field table definition file is located so that when the application is started, the location of field definition files is passed to the Notification Service servers (see Listing 7-3).

In Listing 7-1, the code that is shown in **bold** text shows how the data filtering is implemented in an event poster application. Only subscriptions that contain the name/value pair `billing` and `patient_account` will receive the event.

Listing 7-1 Sample Data Filtering Using the Oracle Simple Events API (C++)

```
CosNotification::StructuredEvent notif;
notif.header.fixed_header.event_type.domain_name =
       CORBA::string_dup("HEALTHCARE");
notif.header.fixed_header.event_type.type_name =
       CORBA::string_dup("HMO");
// Specify an additional filter, based upon name and value
// for this event.
notif.filterable_data.length(2);
notif.filterable_data[0].name = CORBA::string_dup("billing");
notif.filterable_data[0].value <<= CORBA::Long(1999);
notif.filterable_data[1].name =
          CORBA::string_dup("patient_account");
notif.filterable_data[1].value <<= CORBA::Long(2345);
// Push the structured event onto the channel.
testChannel->push_structured_event(notif);
```

Listing 7-2 shows the FML field table definitions file needed to use data filtering.

Listing 7-2 Data Filtering FML Field Table File

```
*base 2000
#Field Name      Field #  Field Type    Flags     Comments
#-----------     -------  ----------    ------    --------
billing          1        long          -         -
patient_account  2        long          -         -
```

Listing 7-3 shows the content of environment variable file (`envfile`). The `envfile` contains the location of the FML field definitions file.

Note: You can name the environment variable file whatever you want, but the name used must match the name specified for the ENVFILE configuration option `n`, the `SERVERS` section of the `UBBCONFIG` file.

Listing 7-3 Envfile Specification for Data Filtering (envfile) (Microsoft Windows)

```
FLDTBLDIR32=D:\tuxdir\EVENTS_Samples\ADVANCED_Simple_cxx\common
FIELDTBLS32=news_flds
```

Listing 7-4 shows, in **bold** text, how the location of the FML field table file is specified in the `UBBCONFIG` file for the Advanced samples.

Listing 7-4 Specifying the FML Field Definitions File in the UBBCONFIG File

```
*SERVERS
TMSYSEVT
  SRVGRP   = NTS_GRP
  SRVID    = 1
TMUSREVT
  SRVGRP   = NTS_GRP>>$@
  SRVID    = 2
  ENVFILE = "D:\tuxdir\EVENTS_Samples\ADVANCED_Simple_CXX\envfile"
TMNTS
  SRVGRP = NTS_GRP
  SRVID = 3
   ENVFILE = "D:\tuxdir\EVENTS_Samples\ADVANCED_Simple_CXX\envfile"
  CLOPT = "-A -- -s TMNTSQS"
TMNTSFWD_T
  SRVGRP = NTS_GRP
  SRVID = 4
  ENVFILE = "D:\tuxdir\EVENTS_Samples\ADVANCED_Simple_CXX\envfile"
TMNTSFWD_P
  SRVGRP = NTS_GRP
```

```
SRVID = 5
ENVFILE = "D:\tuxdir\EVENTS_Samples\ADVANCED_Simple_CXX\envfile"
```

Setting the Host and Port

The object references host and port number requirements for the callback object are as follows:

- For transient callback objects, any port is sufficient and can be obtained dynamically by the ORB.
- For persistent callback objects, the ORB must be configured to accept requests for the callback object on the same port on which the object reference for the callback object was created.

You specify the port number from the user range of port numbers, rather than from the dynamic range. Assigning port numbers from the user range prevents joint client/server applications from using conflicting ports.

The method you use to set the host and port depends on the programming language you are using.

- Setting Host and Port on C++ Subscriber Applications

  For C++ subscriber applications, to specify a particular port for the joint client/server application to use, include the following on the command line that starts the process for the joint client/server application:

  ```
  -ORBport nnnn  -IRBid BEA_IIOP
  ```

  where *nnnn* is the number of the port to be used by the ORB when creating invocations and listening for invocations on the callback object in the joint client/server application.

  Use this command when you want the object reference for the callback object in a joint client/server application to be persistent and when you want to stop and restart the joint client/server application. If this command is not used, the ORB uses a random port. If a random port is used when the joint client/server application is stopped and then restarted, invocations to persistent callback objects in the joint client/server application will fail.

  The port number is part of the input to the `argv` argument of the `CORBA::orb_init` member function. When the `argv` argument is passed, the ORB reads that information, establishing the port for any object references created in that process.

Creating a Transaction Log

When you use persistent subscriptions, you must configure and boot the Oracle Tuxedo queuing system. The queuing system requires a transaction log. Listing 7-5 shows how to use the `tmadmin` utility to create a transaction log.

Listing 7-5 Creating a Transaction Log (createtlog) (Microsoft Windows)

```
>tmadmin
>crdl -b 100 -z D:\tuxdir\EVENTS_Samples\ADVANCED_Simple_CXX\TLOG
>crlog -m SITE1
>quit
>
```

Creating Event Queues

When you use persistent events, you must configure and boot the Oracle Tuxedo queuing system. Two event queues must be created:

- `TMNTSFWD_P`

  This is the event forwarding queue for persistent subscriptions. Events go to this queue first and then are forwarded to matching persistent subscriptions. If an event cannot be delivered on the first attempt, it is held in this queue and repeated attempts are made to deliver it. If the settable retry limit is reached before the event can be successfully delivered, the event is moved to the error queue.

  This queue requires the following configuration parameters:

  – Queuing order (for example, first in, first out).

  – How to handle out-of-order enqueuing.

  – Retry limit (how many retries before moving the event to the error queue).

  – Retry time interval.

  – How full the queue can get before administrative intervention is required.

  – How low the queue can get after getting full before administrative intervention is required.

  - Definition of the administrative intervention command.

- TMNTSFWD_E

  This is the error queue. This queue receives events from the TMNTSFWD_P queue that cannot be delivered to subscriptions. This queue requires the same configuration parameters as the TMNTSFWD_P forwarding queue, however, the retry limit and retry time interval parameters are irrelevant because this is the error queue and errors are only removed by administrative intervention.

To configure these queues, perform the following steps:

1. Create a device on disk for the queue space.
2. Configure a queue space.
3. Create the queues.

These steps are described in the following sections.

Determining Space Parameters for Transient and Persistent Subscriptions

To tune your system for maximum performance, you should determine the optimal values for the following parameters:

- The number of transient forwarding servers (TMNTSFWD_T) and persistent forwarding servers (TMNTSFWD_P).
- IPC queue space (this is used for transient subscriptions).
- Size of /Q queues (this is used for persistent subscriptions).

IPC Queue Space for Transient Subscriptions

Proceed as follows to determine space parameters for transient subscriptions:

1. Determine how many events may be in the pipeline for transient subscriptions; that is, how many events may be in the process of being delivered at any given time. This equals the number of events multiplied by the number of subscribers receiving them.
2. Determine the size of your events. For purposes of this discussion, we will assume that they are relatively small—about 300 bytes or less.

3. Determine how many transient forwarding servers you would like to start, most likely one or two—one per processor on your machine is a good number to start with.

4. Determine how much IPC queue space you will need to hold your transient events. The amount of space you need is 1000 bytes multiplied by the number of events you allow in the pipeline. Divide this number by the number IPC queues your transient forwarders have. If you use MSSQ sets, then your transient forwarders share one IPC queue; if you do not, then each forwarder has its own IPC queue.

   For example, if you estimate that there will be 10 events delivered to 50 subscribers in the pipeline, and you start 2 transient forwarders and they do not share an IPC queue (that is, you do not use MSSQ sets), the amount of IPC queue space you need is:

   10 events * 50 subscribers * 1000 bytes / 2 forwarders = 250,000 bytes

5. Configure the IPC queue size to that number by changing the entries in the system registry. How you do this is platform-specific.

   – For Microsoft Windows systems, see “Setting IPC Parameters on Microsoft Windows” on page -12.

   – For UNIX systems, refer to the system reference manual supplied with the system.

/Q Queue Size Parameter Persistent Subscriptions

Proceed as follows to determine space parameters for persistent subscriptions:

1. Determine how many events may be in the pipeline for persistent subscriptions; that is, how many events may be in the process of being delivered at any given time. This equals the number of events multiplied by the number of subscribers receiving them.

2. Determine the size of your events. For purposes of this discussion, we will assume that they are relatively small—about 300 bytes or less.

3. Determine the size your /Q queues need to be to hold your persistent events (both for your pending queue and error queue). Proceed as follows to do this:

   a. Determine the size of a disk page. This is platform-specific. For example, on Microsoft Windows, a disk page is 500 bytes. On UNIX machines, a disk page could range from 500 to 4000 bytes in size.

   b. Determine how many disk pages you will need to store one event rounding up. For example, if you need 1000 bytes per event and disk pages are 500 bytes, you will need 2 disk pages per event.

c. Determine how many disk pages you will need for your events. For example, if you want to allow 500 pending events and 200 error events, and an event takes up 2 disk pages, you will need 1400 disk pages.

d. Determine how many disk pages you will need for your qspace. This is the number of disk pages you need for your events plus some pages for qspace overhead. For example, if you need 1400 disk pages for events, then your qspace needs approximately 1450 disk pages (50 pages of qspace overhead).

e. Determine how many pages you will need for your qspace device. This is the number of pages you need for the qspace plus some pages for device overhead. For example, if you need 1450 disk pages for your qspace, then your device needs approximately 1500 pages (50 pages of device overhead).

4. When you use `qmadmin` to create the qspace for your persistent events, the first phase is to create a device. Use the size computed above in step 3e above (approximately 1500 pages). Next, specify the size of the qspace. Use the size computed in step 3d (approximately 1450 pages). Next, specify how many events can be in the pending queue and how many events can be in the error queue. The following sections explain how to create and configure qspaces.

Creating a Device on Disk for the Queue Space

You use the qmadmin command utility to create a device on disk for the queue space.

Before you create a queue space, you must create an entry for it in the universal device list (UDL). Listing 7-6 shows an example of the commands.

Listing 7-6 Creating a Device on Disk for Queue Space (UNIX)

```
prompt>qmadmin d:\smith\reg\QUE
qmadmin - Copyright (c) 1996-1999 BEA Systems, Inc.
Portions * Copyright 1986-1997 RSA Data Security, Inc.
All Rights Reserved.
Distributed under license by BEA Systems, Inc.
Oracle Tuxedo is a registered trademark.
QMCONFIG=d:\smith\reg\QUE
> crdl d:\smith\reg\QUE 0 1100
Created device d:\smith\reg\QUE, offset 0, size 1100
 on d:\smith\reg\QUE
```

For more information about creating a device on disk, see *Using the ATMI /Q Component*.

Configuring a Queue Space

You use the `qmdamin qspacecreate` command to configure queue spaces. A queue space makes use of IPC resources; therefore, when you define a queue space you are allocating a shared memory segment and a semaphore. The easiest way to use the `qspacecreate` command is to let it prompt you. Listing 7-7 shows an example queue space that is configured for the Advanced sample application.

Listing 7-7 Creating Queue Space

```
> qspacecreate
Queue space name: TMNTSQS
IPC Key for queue space: 52359
Size of queue space in disk pages: 1050
Number of queues in queue space: 2
Number of concurrent transactions in queue space: 10
Number of concurrent processes in queue space: 10
Number of messages in queue space: 500
Error queue name: TMNTSFWD_E
Initialize extents (y, n [default=n]): y
Blocking factor [default=16]:
```

In the queue space created in Listing 7-7, take note of the following size settings:

`Number of messages in queue space:500`
: Setting this parameter to 500 allows room for a total of 500 events in the pending and error queues.

`Size of queue space in disk pages:1050`
: On Microsoft Windows, each disk page is 500 bytes and each event needs 1000 bytes. In addition, you must allow 2 disk pages per event. Since you estimate that there will be 500 events in the pending and error queues, then you must allow 1000 disk pages to store them (500 * 2). Also, you must allow 50 disk pages for qspace overhead, so the qspace size is set to 1050 disk pages. Finally, the device needs 50 disk pages of overhead too, so the device size is 1100 disk pages, which you set using the `crdl` command (see Listing 7-6).

For more information about creating queue space, see *Using the ATMI /Q Component*.

Creating the Queues

You must use the `qmadmin qcreate` command to create each queue that you intend to use. Before you can create a queue, you first have to open the queue space with the `qmadmin qopen` command. If you do not provide a queue space name, `qopen` will prompt for it.

Listing 7-8 shows an example of creating the `TMNTSFWD_P` and `TMNTSFWD_E` queues that are created for the Advanced sample application.

Listing 7-8 Creating Queues

```
> qopen
Queue space name: TMNTSQS

> qcreate
Queue name: TMNTSFWD_P
Queue order (priority, time, fifo, lifo): fifo
Out-of-ordering enqueuing (top, msgid, [default=none]): none
Retries [default=0]: 5
Retry delay in seconds [default=0]: 3
High limit for queue capacity warning (b for bytes used, B for
 blocks used, % for percent used, m for messages [default=100%]):
 80%
Reset (low) limit for queue capacity warning [default=0%]: 0%
Queue capacity command:
No default queue capacity command
Queue 'TMNTSFWD_P' created

> qcreate
Queue name: TMNTSFWD_E
Queue order (priority, time, fifo, lifo): fifo
Out-of-ordering enqueuing (top, msgid, [default=none]): none
Retries [default=0]: 2
Retry delay in seconds [default=0]: 30
High limit for queue capacity warning (b for bytes used, B for
 blocks used, % for percent used, m for messages [default=100%]):
 80%
```

```
Reset (low) limit for queue capacity warning [default=0%]: 0%
Queue capacity command:
No default queue capacity command
Q_CAT:1438: INFO: Create queue - error queue TMNTSFWD_E created
Queue 'TMNTSFWD_E' created

> q
```

For more information about creating queues, see *Using the ATMI /Q Component*.

Setting IPC Parameters on Microsoft Windows

The Oracle Tuxedo software for Microsoft Windows systems provides you with Oracle Tuxedo IPC Helper (TUXIPC), an interprocess communication subsystem, that is installed with the product. On most machines, IPC Helper runs as installed; however, you can use the IPC Resources page of the control panel applet to tune the TUXIPC subsystem and maximize performance.

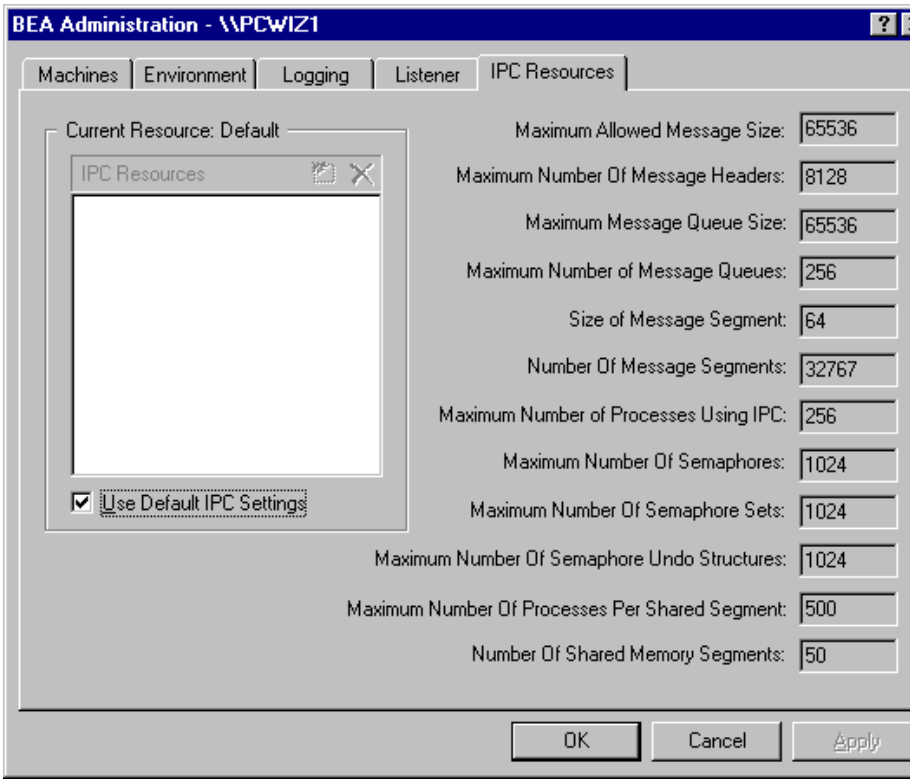
To display the IPC Resources page of the IPC Control Panel, perform these steps:

1. Click Start—>Settings—>Control Panel. The Microsoft Windows Control Panel is displayed (Figure 7-1).

Figure 7-1 Microsoft Windows Control Panel

2. Click the Oracle Administration icon. The Oracle Administration Control Panel is displayed (Figure 7-2).
3. Click on the IPC Resources tab. The IPC Resources Control Panel portion of the Oracle Administration Control Panel is displayed (Figure 7-2).

Figure 7-2 Oracle Tuxedo Software for Microsoft Windows IPC Resources Control Panel



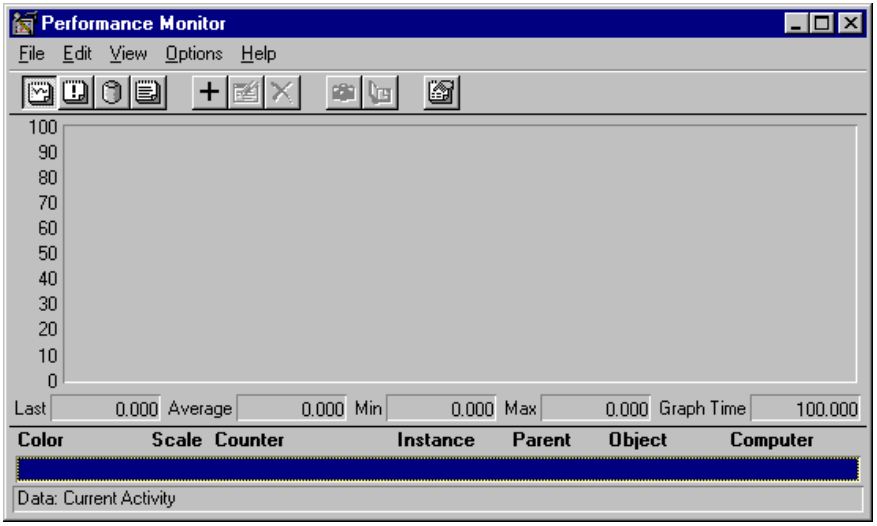
To define IPC settings for your Oracle Tuxedo machine, proceed as follows:

1. In the Current Resource Default box, click the Use Default IPC Settings check box to clear it.
2. Click the insert box.
3. Enter the name of your machine and press Enter.
4. Click the fields next to the IPC resources you want to set, enter the desired values, and click Apply. Clicking Apply saves the changes in the Registry Table. You must then stop and then restart the `tuxipc.exe` service for the changes to take effect.
5. Click OK to close the Control Panel.

You can view the performance of a running Oracle Tuxedo server application on the Performance Monitor.

To start the Performance Monitor, click Start—>Programs—>Administration Tools—>Performance Monitor on the taskbar. The Performance Monitor screen is displayed (Figure 7-3).

Figure 7-3 Oracle Tuxedo Software for Microsoft Windows Performance Monitor



Creating the UBBCONFIG File and the TUXCONFIG File

For event poster and subscriber applications to communicate with a CORBA object in the Oracle Tuxedo domain, in this case the Notification Service, a `UBBCONFIG` file is required for the Notification Service. The `UBBCONFIG` file must be written as part of the development of the Notification Service application; otherwise, you will not be able to build and run the application.

After you write the `UBBCONFIG` file, you use the `tmloadcf` command to produce the `TUXCONFIG` file, which is used at run time. Therefore, the `TUXCONFIG` file must exist before the Notification Service application is started. The `TUXCONFIG` file is simply a binary version of the `UBBCONFIG` file. The following is an example of how to use the tmloadcf command:

```
tmloadcf -y ubb
```

Before writing the `UBBCONFIG`, you should list the configuration requirements of your Notification Service application. To list requirements, determine the required servers and processes to support the subscription. Table 7-1 shows the configuration requirements for the different types of subscriptions.

Table 7-1 Configuration Requirements for Transient and Persistent Subscriptions

To support these types of subscriptions	Your UBBCONFIG file must include the following servers, and processes
Transient subscription	TMUSREVT, TMNTS, and TMNTSFWD_T
Persistent subscription	TMUSREVT, TMNTS, TMNTSFWD_P, TMQUEUE, TMQFORWARD

If you are using event subscriber applications that use IIOP, you need to configure the IIOP Listener (ISL) command in the UBBCONFIG file with parameters that enable outbound IIOP to invoke callback objects that are not connected to an IIOP Handler (ISH). The -O option (uppercase letter O) of the ISL command enables outbound IIOP. Additional parameters allow system administrators to obtain the optimum configuration for their Notification Service application. For more information about the ISL command, see *Setting Up an Oracle Tuxedo Application*.

When developing a Notification Service application, the SERVERS section of the UBBCONFIG file may include the following types of servers:

- TMUSREVT

  An Oracle Tuxedo system-provided server that processes event report message buffers from tppost(3), and acts as an EventBroker to filter and distribute them. (Required)

- TMNTS

  An Oracle Tuxedo Notification Service server that processes requests for subscriptions and event postings. (Required)

- TMNTSFWD_T

  An Oracle Tuxedo Notification Service server that forwards transient events to subscribers of transient subscriptions. (Required for transient subscriptions)

- TMNTSFWD_P

  An Oracle Tuxedo Notification Service server that forwards persistent events to subscribers that have persistent subscriptions. Events that cannot be delivered to subscribers are sent to the error queue. (Required for persistent subscriptions)

- TMQUEUE

An Oracle Tuxedo server that manages event queues. (Required for persistent subscriptions)

- `TMQFORWARD`

  An Oracle Tuxedo server that forwards events to the Notification Service `TMNTSFWD_P` server so that they can be forwarded to persistent subscribers. (Required for persistent subscriptions)

- `ISL`

  The Oracle Tuxedo IIOP Server Listener/Handler process. (Required if the event poster or subscriber is remote, that is outside the local domain)

The `UBBCONFIG` file shown in Listing 7-9 is from the Notification Service Introductory sample application. The Introductory sample application supports transient subscriptions only; it does not support persistent subscriptions or data filtering.

Listing 7-9 The Introductory Sample UBBCONFIG File

```
# This UBBCONFIG file supports transient subscriptions only; it does
# not persistent subscriptions or data filtering.
*RESOURCES
    IPCKEY 52359
    DOMAINID events_intro_simple_cxx
    MASTER SITE1
    MODEL SHM
#--------------------------------------------------------------------
*MACHINES
    "BEANIE"
        LMID = SITE1
    APPDIR = "D:\tuxdir\EVENTS~1\INTROD~2"
    TUXCONFIG = "D:\tuxdir\EVENTS~1\INTROD~2\tuxconfig"
    TUXDIR = "d:\tuxdir"
    MAXWSCLIENTS = 10
    ULOGPFX = "D:\tuxdir\EVENTS~1\INTROD~2\ULOG"
#--------------------------------------------------------------------
# Since we are using transient events, the group need not be
# transactional.
*GROUPS
    SYS_GRP
```

```
    LMID = SITE1
    GRPNO = 1
#------------------------------------------------------------------
*SERVERS
    DEFAULT:
    CLOPT = "-A"
    TMSYSEVT
    SRVGRP = SYS_GRP
    SRVID = 1
TMUSREVT
    SRVGRP = SYS_GRP
    SRVID = 2
TMFFNAME
    SRVGRP = SYS_GRP
    SRVID = 3
    CLOPT = "-A -- -N -M"
TMFFNAME
    SRVGRP = SYS_GRP
    SRVID = 4
    CLOPT = "-A -- -N"
TMFFNAME
    SRVGRP = SYS_GRP
    SRVID = 5
    CLOPT = "-A -- -F"
# Start the notification service server.
#
TMNTS
    SRVGRP = SYS_GRP
    SRVID = 6
# Start the Notification Service transient event forwarder.
#
TMNTSFWD_T
    SRVGRP = SYS_GRP
    SRVID = 7
# Start the ISL with -O since we are using callbacks to clients.
ISL
    SRVGRP = SYS_GRP
    SRVID = 8
```

```
    CLOPT = "-A -- -O -n //BEANIE:2359"
#--------------------------------------------------------------------
*SERVICES
```

The code example shown in Listing 7-10 is from the Notification Service Advanced sample application. The Advanced sample application supports transient and persistent subscriptions and data filtering.

Listing 7-10 The Advanced Sample UBBCONFIG File

```
# This UBBCONFIG file supports transient and persistent
# subscriptions and data filtering.
*RESOURCES
    IPCKEY 52363
    DOMAINID events_advanced_simple_cxx
    MASTER SITE1
    MODEL SHM
#--------------------------------------------------------------------
*MACHINES
    "BEANIE"
        LMID = SITE1
    APPDIR = "D:\tuxdir\EVENTS~1\ADVANC~1"
    TUXCONFIG = "D:\tuxdir\EVENTS~1\ADVANC~1\tuxconfig"
    TUXDIR = "d:\tuxdir"
    MAXWSCLIENTS = 10
    ULOGPFX = "D:\tuxdir\EVENTS~1\ADVANC~1\ULOG"
#
#  Since we are using persistent events, we need a transaction log.
#
    TLOGDEVICE = "D:\tuxdir\EVENTS~1\ADVANC~1\TLOG"
    TLOGSIZE = 10
#--------------------------------------------------------------------
*GROUPS
    SYS_GRP
    LMID = SITE1
    GRPNO = 1
```

```
#  Create a null transactional group for the notification service
#  servers.
#
NTS_GRP
    LMID = SITE1
    GRPNO = 2
    TMSNAME = TMS
    TMSCOUNT = 2
#  Since we are using persistent events, we need a persistent queue
#  create a queue transactional group for the queue servers.
#
QUE_GRP
    LMID = SITE1
    GRPNO = 3
    TMSNAME = TMS_QM
    TMSCOUNT = 2
#
#  Make the queue group manage the QUE space we create.
#  The name of the queue space specified here as TMNTSQS must match #  the
name of the queue space you created.
#
   OPENINFO = "TUXEDO/QM:D:\tuxdir\EVENTS~1\ADVANC~1\QUE;TMNTSQS"
#-------------------------------------------------------------------
*SERVERS
    DEFAULT:
    CLOPT = "-A"
#
#  Start the queue server.
#  The name of the queue space specified in the -s option of
#  CLOPT must match the name of the queue space you created.
#
TMQUEUE
    SRVGRP = QUE_GRP
    SRVID = 1
    CLOPT = "-s TMNTSQS:TMQUEUE --  "
#
#  Start the queue forwarder, have it forward events to the
#  notification service persistent forwarder.
```

```
#
TMQFORWARD
    SRVGRP = QUE_GRP
    SRVID = 2
    CLOPT = "-- -i 2 -q TMNTSFWD_P"
    TMSYSEVT
    SRVGRP = NTS_GRP
    SRVID = 1
#
#  Start the user EventBroker. Pass in the environment file
#  so that the user EventBroker can find the "Story" fml field
#  definition. This allows the user EventBroker to perform
#  data filtering.
#
TMUSREVT
    SRVGRP = NTS_GRP
    SRVID = 2
    ENVFILE = "D:\tuxdir\EVENTS~1\ADVANC~1\envfile"
TMFFNAME
    SRVGRP = SYS_GRP
    SRVID = 1
    CLOPT = "-A -- -N -M"
TMFFNAME
    SRVGRP = SYS_GRP
    SRVID = 2
    CLOPT = "-A -- -N"
TMFFNAME
    SRVGRP = SYS_GRP
    SRVID = 3
    CLOPT = "-A -- -F"
#
#  Start the notification service server. Pass in the environment
#  file so that the notification server can perform data filtering.
#  The -s option must be specified since we are using
#  persistent events. Note that the -s option specifies the name
#  of the queue space as TMNTSQS. This name must match the name
#  of the queue space you created.
#
```

```
TMNTS
    SRVGRP = NTS_GRP
    SRVID = 3
    ENVFILE = "D:\tuxdir\EVENTS~1\ADVANC~1\envfile"
    CLOPT = "-A -- -s TMNTSQS"
#
#  Start the notification service transient event forwarder.
#  Pass in the environment file so that the server can perform
#  data filtering.
#
TMNTSFWD_T
    SRVGRP = NTS_GRP
    SRVID = 4
    ENVFILE = "D:\tuxdir\EVENTS~1\ADVANC~1\envfile"
#
#  Start the notification service persistent event forwarder.
#  Pass in the environment file so that the server can perform
#  data filtering.
#
TMNTSFWD_P
    SRVGRP = NTS_GRP
    SRVID = 5
    ENVFILE = "D:\tuxdir\EVENTS~1\ADVANC~1\envfile"
#
#  Start the ISL with -O since we're using callbacks to clients.
#
ISL
    SRVGRP = SYS_GRP
    SRVID = 4
    CLOPT = "-A -- -O -n //BEANIE:2363"
#--------------------------------------------------------------
*SERVICES
```

Managing the Notification Service

After you have deployed the Notification Service application, you may need to perform the following administrative tasks on an on-going basis:

1. Synchronize databases.
2. Purge the system of dead subscriptions.
3. Monitor queue utilization.
4. Purge the queues of unwanted events.
5. Move or remove events from the error queue.

Synchronizing Databases

If you configure more than one EventBroker, then your Notification Service subscription databases will have to be synchronized. Because the synchronization process requires time—time that can impact event delivery—and increases network traffic, you should not configure more than one EventBroker unless the event traffic warrants it.

When you configure more than one EventBroker, you can configure time required to synchronize the databases using the `-P` option on the TMUSREVT server. For more information on how to set this option, see `TMUSREVT(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.

Note: The time required to synchronize the databases affects the elapsed time from when a subscriber subscribes and when it receives events. It also affects the elapsed time from when a subscriber unsubscribes and when it stops receiving events.

Purging the System of Dead Subscriptions

A subscription dies in one of two ways: (1) the subscriber creates a persistent subscription, shuts down without unsubscribing, and then does not restart and reconnect to the Notification Service, or, (2) the subscriber creates a subscription that never matches any event. While it is allowable for a subscriber to create a persistent subscription and then shut down without unsubscribing, it is an error if the subscriber does not periodically reconnect for the purpose of picking up accumulated events. Because the Notification Service periodically attempts to deliver events that match persistent subscriptions, such events accumulate while the subscriber is disconnected, consume queue space, and waste system resources.

Subscriptions that will never match any events should not be created because they serve no useful purpose. Also, subscriptions consume system resources because each posted event must be compared against each subscription.

Using the `ntsadmin` commands listed in Table 7-2, you can view all subscriptions and see how many events are currently in the pending queue and in the error queue for each subscription. You can also remove subscriptions using a `ntsadmin` command or move events from the error queue to the pending queue. For a description of the `ntsadmin` utility, see "ntsadmin" on page -26.

Table 7-2 ntsadmin Commands Summary

Command	Usage
`subscriptions`	Lists subscriptions in the subscription database.
`rmsubscriptions`	Removes subscriptions for the subscription database.
`pendevents`	Lists information about events in the pending events queue. (For persistent subscriptions only.)
`rmpendevents`	Removes events in the pending events queue. (For persistent subscriptions only.)
`errevents`	Lists events in the event error queue. (For persistent subscriptions only.)
`rmerrevents`	Removes events in the events error queue. (For persistent subscriptions only.)

Although there is no way of automatically detecting a dead subscription, the `ntsadmin` utility is helpful in determining when and if a subscription is dead.

Monitoring Queue Utilization

Queues are created with a fixed amount of space allocated to them. This space is consumed as events accumulate in the queues. If the queues become full, subsequent attempts to enqueue events will fail.

You use `qmadmin` or `ntsadmin` to monitor queue utilization (see `qmadmin(1)` in the *Oracle Tuxedo Command Reference*).

When the queue space was created to hold the pending events, the maximum number of events that could be held by the queue space was specified. For example, in the Advanced sample

application, the maximum number of events for the TMNTSQS queue space was set to 200 (see "Creating Event Queues" on page -6). With knowledge of queue space capacity, you can use the ntsadmin pendevents command to determine the number of events pending in the event queue. If the event queue is full or nearly full, you may want to increase the setting for maximum number of events or increase the number of event queues.

Note: Use the threshold command option (cmd) on the qmadmin qcreate command to generate a warning when a queue is nearing capacity. For information on this command, see qmadmin(1) in the *Oracle Tuxedo Command Reference*.

Purging the Queues of Unwanted Events

You can purge events from either the pending queue or the error queue by using the ntsadmin commands rmerrevents and rmpendevents.

WARNING: After an event has been removed from the queue there is no way to recover it. The event is gone and the subscribing application will never receive the event.

Managing the Error Queue

After a preset number of attempts to deliver an event, the event is moved to the error queue. Once on the error queue, the administrator must take some action to either purge the event from the system, or move the event from the error queue back to the pending queue. Purging of events is discussed in the previous section.

When you move an event from the error queue back to the pending queue, you are requesting that the system resume delivery attempts of the event. Because failed attempts to deliver events consume system resources, you should not do this unless you have some reason to believe that the condition that prevented delivery before has been corrected. The ntsadmin retryerrevents command is provided specifically to move events back to the pending queue.

Notification Service Administration Utility and Commands

This topic includes the following sections:

ntsadmin Utility

This section describes the `ntsadmin` utility.

ntsadmin

Synopsis

Oracle Tuxedo CORBA Notification Service administration command interpreter.

Syntax

```
ntsadmin
```

Description

The Notification Service includes an administration command interpreter, `ntsadmin`, that provides commands to perform the following tasks for CORBA Notification Service applications:

- List subscriptions
- Delete subscriptions
- Display summary information about structured events on the pending and error queues
- Delete structured events on the pending and error queues
- Move structured events from the error queue to the pending queue

Note: When you enter `ntsadmin` to start the program, if your application only has transient subscriptions, the commands for persistent subscriptions are disabled.

Note: The Notification Service must be running before you can use `ntsadmin`.

You can exit the `ntsadmin` program by entering a `q` (for quit) at the command prompt. You can terminate the output from a command by pressing the Break key; the program then prompts for a new command.

Output from `ntsadmin` is paginated according to the pagination command in use (see the `paginate` command).

Note: The `subscription` command has different output depending on the setting of the `verbose` command.

Security

This utility can only be used by the system administrator.

See Also

`TMNTS, TMNTSFWD_T, TMNTSFWD_P, qmadmin`

ntsadmin Commands

Commands may be entered either by their full name or by an abbreviation (if available, the abbreviation is listed below in parentheses following the full name), followed by appropriate arguments. Arguments that appear in square brackets [] are optional; arguments in curly braces {} indicate a selection from mutually exclusive options. Each command offers the following options:

Option	Definition
`[ -i identifier ]`	If specified, identifies the subscription that matches `identifier`.
`[ -n name ]`	If specified, identifies the subscription(s) with a subscription name that matches ***`name`*** only. To specify names which match the empty string (that is, subscriptions with no name), enclose an empty string between quotes (" "). **Note:** This option does not support the wildcard character (*) so name must match the subscription name exactly.
`[ -t ]`	If specified, designates subscriptions with a QoS of transient only.
`[ -p ]`	If specified, designates subscriptions with a QoS of persistent only.

The `ntsadmin` commands are as follows:

`subscriptions (sub) [{-i identifier |-n name |-t | -p}]`
Lists subscriptions in the subscription database.

Note: The `subscription` command has different output depending on whether the verbose mode is on or off (the `verbose` command is described below). Listing 7-11 shows examples of `subscription` output with verbose on and off.

Listing 7-11 Subscription Command Output with Verbose Mode On and Off

```
> verbose on
Verbose mode is now on
> sub
          ID: 1000000006
        Name: marcello
          QoS: Transient
      Qspace: <N/A>
  Expression: stock trade\.quote
      Filter: stock_name %% 'BEAS' && price_per_share > 150
          ID: 1000000005
        Name: marcello
         QoS: Persistent
      Qspace: TMNTSQS
  Expression: stock trade\.sell
      Filter:
          ID: 1000000004
        Name: marcello
         QoS: Persistent
      Qspace: TMNTSQS
  Expression: stock trade\.buy
      Filter:
> verbose off
Verbose mode is now off
> sub
ID          Name                 Expression
--          ----                 ----------
1000000006  marcello         [T] stock trade\.quote
1000000005  marcello         [P] stock trade\.sell
1000000004  marcello         [P] stock trade\.buy
```

rmsubscriptions (rmsub) [{-i *identifier* |-n *name* |-t | -p]}[-y]
: Removes subscriptions from the subscription database. This command prompts for confirmation unless **-y** is used.

This command displays the number of subscriptions removed.

pendevents (pevt) [{-i *identifier* |-n *name*}]
Lists information about events in the pending events queue.

rmpendevents (rmpevt) [{-i *identifier* |-n *name* |-o}][-y]
Removes events in the pending events queue. If **-o** is specified, all events that do not currently have a corresponding subscription in the subscription database will be removed.

This command prompts for confirmation unless **-y** is used and displays the number of events removed.

errevents (eevt) [{-i *identifier* |-n *name*}]
Lists events in the events error queue.

rmerrevents (rmeevt) [{-i *identifier* |-n *name* |-o}][-y]
Removes events in the events error queue. If **-o** is specified, all events that do not currently have a corresponding subscription in the subscription database will be removed.

This command prompts for confirmation unless **-y** is used and displays the number of events removed.

retryerrevents (reteevt) [{-i *identifier* |-n *name*}][-y]
Retries the events in the events error queue. This will move the events from the error queue to the pending queue.

This command prompts for confirmation unless **-y** is used and displays the number of events moved from the error queue to the pending queue.

quit (q)
Terminates the session.

echo (e) [{off |on}]
Echoes input command lines when set to **on**. If no input is given, then the current setting is toggled and the new setting is printed. The initial setting is **off**.

help (h) [{*command* |all}]
Prints help messages. If ***command*** is specified, the abbreviation, arguments and description for that command are printed. **all** causes a description of the commands to be displayed. Omitting all arguments causes the syntax of all commands to be displayed.

paginate (page) [{off |on}]
Paginates output. If no input is given, the current setting is toggled and the new setting is printed. The initial setting is **on**, unless either standard input or standard output is a non-terminal device. Pagination may only be turned on when both standard input and standard output are terminal devices. The shell environment variable PAGER may be used

to override the default command used for paging output. The default paging command is the pager indigenous to the native operating system environment; for example, the command `pg` is the default on UNIX operating systems.

`verbose (v) [{on | off }]`
Produces output in verbose mode. If no option is given then the current setting will be toggled, and the setting is printed. The initial setting is **`off`**.

`! shellcommand`
Use this command to escape to shell and execute ***`shellcommand`***.

`!!`
Use this command to repeat the previous shell command.

`#[text]`
Use this command to designate the line as a comment.

`<CR>`
Use this command to repeat the previous command.

Using the ntsadmin Utility

This section provides examples of using the `ntsadmin` utility.

Listing 7-12 shows an example of using `ntsadmin` to move events from the error queue back to the pending queue. The following steps are performed:

1. Look up all subscriptions for `marcello`.
2. Use the unique `subscription_id` to display information about events on the error queue.
3. Move the events from the error queue to the pending queue.

Listing 7-12 Moving Events from the Error Queue to the Pending Queue

```
D:\smith\reg>ntsadmin
ntsadmin - Copyright (c) 1996-1999 BEA Systems, Inc.
Portions * Copyright 1986-1997 RSA Data Security, Inc.
All Rights Reserved.
Distributed under license by BEA Systems, Inc.
Oracle Tuxedo is a registered trademark.
INFO: /Q Qspace - TMNTSQS
INFO: /Q Device - D:\smith\reg\QUE (SITE1)
> subscriptions -n marcello
```

```
ID          Name                Expression
--          ----                ----------
1000000002  marcello         [T] stock trade\.quote
1000000001  marcello         [P] stock trade\.sell
1000000000  marcello         [P] stock trade\.buy

> verbose off
Verbose mode is now off

> eevt -i 1000000003
ID          Name            Count
--          ----            -----
1000000003  marcello        1
> reteevt -i 1000000003 -y
1 event(s) retried
```

Listing 7-13 shows an example of using `ntsadmin` to remove subscriptions and purge events.

Listing 7-13 Removing a Subscription

```
> rmsub -n BillJones -y
2 subscription(s) removed
> rmeevt -n marcello -y
1 event(s) removed
> rmpevt -n BillJones -y
No events removed
```

Listing 7-14 shows how to check events pending for a specific subscription.

Listing 7-14 Checking for Pending Events

```
> pevt -n marcello
ID          Name            Count
--          ----            -----
1000000003  marcello         1
```

Notification Servers

This section provides descriptions of the following servers:

- TMTNS
- TMNTSFWD_T
- TMNTSFWD_P

The Notification Service also uses the following Oracle Tuxedo system servers. For descriptions of these servers, refer to the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.

- TMSYSEVT(5)
- TMUSREVT(5)
- TMQFORWARD(5)
- TMQUEUE(5)

TMNTS

Synopsis

Processes requests for subscriptions and event postings.

Syntax

```
TMNTS SRVGRP="identifier" SRVID="number"
      [CLOPT="[-A] [servopts options]
      [--[-S queuespace]"]
```

Description

TMNTS is an Oracle Tuxedo-provided server that processes all requests for subscriptions and event postings.

Parameter

-S queuespace

The name of the queue space to use. This queue space must contain two queues: TMNTSFWD_P and TMNTSFWD_E. This option is required for persistent subscriptions only.

Note: If you plan to use subscriptions with a QoS of `Persistent`, you must create a queue space, a queue for holding events, and an error queue before the system is operational. The queue space name must match the *queuespace* name specified using the `CLOPT -S` *`queuespace`* parameter for the `TMNTS` server. The event queue must be named `TMNTSFWD_P`. The error queue must be named `TMNTSFWD_E`.

It is possible to boot more then one `TMNTS` server to increase reliability and availability.

The `TMNTS` server must be part of a transactional group if events will be posted in the context of a transaction.

Interoperability

`TMNTS` must run on Oracle WebLogic Enterprise version 5.0 or later or Oracle Tuxedo 8.0 or later.

Notes

The `TMNTS` server relies on services provided by the `TMUSREVT` and `TMSYSEVT` servers. Therefore, these servers must be booted before the system is operational. If transient subscriptions are used, the `TMNTSFWD_T` server must also be booted before the system is operational. If persistent subscriptions are used, the `TMNTSFWD_P`**,** `TMQUEUE`, and `TMQFORWARD` servers must also be booted before the system is operational.

Example

```
*SERVERS

TMNTS SRVGRP = NTS_GRP  SRVID = 3
       CLOPT = "-A -- -s TMNTSQS"
```

See Also

`TMSYSEVT(5)`, `TMUSREVT(5)`, `TMQUEUE(5)`, `TMQFORWARD(5)`, `TMNTSFWD_P`, `TMNTSFWD_T(5)`, `UBBCONFIG(5)`

TMNTSFWD_T

Synopsis

Forwards events to transient subscribers.

Syntax

```
TMNTSFWD_T SRVGRP="identifier" SRVID="number"
       [CLOPT="[-A][--"]
```

Description

TMNTSFWD_T is an Oracle Tuxedo-provided server that forwards events to subscribers who specified a QoS of Transient. There is no transaction context associated with event delivery.

Note: It is possible to boot more then one TMNTSFWD_T server to increase reliability and availability.

Interoperability

TMNTS must run on Oracle WebLogic Enterprise version 5.0 or later or Oracle Tuxedo 8.0 or later.

Notes

The TMNTSFWD_T server relies on services provided by the TMNTS, TMUSREVT, and TMSYSEVT servers. Therefore, these servers must be booted before the system is operational.

Example

```
*SERVERS

TMNTSFWD_T  SRVGRP = SYS_GRP  SRVID = 7
```

See Also

TMSYSEVT(5), TMUSREVT(5), TMNTS(5), TMNTSFWD_P, UBBCONFIG(5). Also, see "IPC Queue Space for Transient Subscriptions" on page -7.

TMNTSFWD_P

Synopsis

Forwards events to persistent subscribers.

Synopsis

```
TMNTSFWD_P SRVGRP="identifier" SRVID="number"
       CLOPT="[-A] [--"]
```

Description

TMNTSFWD_P is an Oracle Tuxedo-provided server that forwards events to subscribers who specified a QoS of persistent. There is no transaction context associated with event delivery.

It is possible to boot more then one TMNTSFWD_P server to increase reliability and availability.

Interoperability

TMNTS must run on Oracle WebLogic Enterprise version 5.0 or later or Oracle Tuxedo 8.0 or later.

Notes

The TMNTSFWD_P server relies on services provided by the TMNTS, TMUSREVT, TMSYSEVT, TMQUEUE, and TMQFORWARD servers. Consequently, these servers must be booted before the system is operational.

This server must be booted in a transactional group.

The number of TMNTSFWD_P servers booted should be the same as the number of TMQFORWARD servers booted.

Example

```
*SERVERS

TMNTSFWD_P  SRVGRP = NTS_GRP  SRVID = 5
```

See Also

TMSYSEVT(5), TMUSREVT(5), TMNTS, TMNTSFWD_T, servopts(5), UBBCONFIG(5)

Index

P

Q

R

S

T

U