

Oracle® Tuxedo

Using the EventBroker

12c Release 2 (12.2.2)

April 2016

ORACLE®

Copyright © 1996, 2016, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

What Is an Event?	1-1
Differences Between Application-defined and System-defined Events.	1-2
What Is the EventBroker?	1-2
How the EventBroker Works	1-3
What Are the Benefits of Brokered Events?	1-6
Process of Using the EventBroker	2-1
How to Configure EventBroker Servers.	2-2
How to Set the Polling Interval.	2-3
Subscribing, Posting, and Unsubscribing to Events with the ATMI and the EVENT_MIB 2-3	
Subscribing, Posting, and Unsubscribing to Events Across Domains	2-6
How to Select a Notification Method	2-9
How to Cancel a Subscription to an Event	2-10
How to Use the EventBroker with Transactions	2-10

About the EventBroker

This topic includes the following sections:

- [What Is an Event?](#)
- [Differences Between Application-defined and System-defined Events](#)
- [What Is the EventBroker?](#)
- [How the EventBroker Works](#)
- [What Are the Benefits of Brokered Events?](#)

What Is an Event?

An event is a state change or other occurrence in a running application (such as a network connection being dropped) that may require intervention by an operator, an administrator, or the software. The Oracle Tuxedo system reports two types of events:

- System-defined events—which are situations (primarily failures) defined by the Oracle Tuxedo system, such as the exceeding of certain system capacity limits, server terminations, security violations, and network failures.
- Application-defined events—which are situations defined by a customer application, such as the ones listed in [Table 1-1](#).

Table 1-1 Application-defined Events

In an application for this type of business . . .	An occurrence of this situation may be defined as an “event” . . .
Stock brokerage	A stock is traded at or above a specified price.
Banking	A withdrawal or deposit above a specified amount is made.
	The cash available in an ATM machine drops below a specified amount.
Manufacturing	An item is out of stock.

Application events are occurrences of application-defined events, and *system events* are occurrences of system-defined events. Both application and system events are received and distributed by the Oracle Tuxedo EventBroker component.

Differences Between Application-defined and System-defined Events

Application-defined events are defined by application designers and are therefore application specific. Any of the events defined for an application may be tracked by the client and server processes running in the application.

System-defined events are defined by the Oracle Tuxedo system code and are generally associated with objects defined in [TM_MIB \(5\)](#). A complete list of system-defined events is published on the [EVENTS \(5\)](#) reference page. Any of these events may be tracked by users of the Oracle Tuxedo system.

The Oracle Tuxedo EventBroker posts both application-defined and system-defined events, and an application can subscribe to events of both types. The two types of events can be distinguished by their names: the names of system-defined events begin with a dot (.); the names of application-specific events cannot begin with a dot (.).

What Is the EventBroker?

The Oracle Tuxedo EventBroker is a tool that provides asynchronous routing of application events among the processes running in a Oracle Tuxedo application. It also distributes system events to whichever application processes want to receive them.

The EventBroker performs the following tasks:

- Monitors events and notifies subscribers when events are posted via `tpost (3c)`.
- Keeps an administrator informed of changes in an application.
- Provides a system-wide summary of events.
- Provides a tool through which an event can trigger a variety of notification activities.
- Provides a filtering capability, providing additional conditions to the posted event's buffer.

Note: For a sample application that you can copy and run as a demo, see “[Tutorial for bankapp, a Full C Application](#)” in *Tutorials for Developing Oracle Tuxedo ATMI Applications*.

The EventBroker recognizes over 100 meaningful state transitions to a MIB object as system events. A posting for a system event includes the current MIB representation of the object on which the event occurred and some event-specific fields that identify the event that occurred. For example, if a machine is partitioned, an event is posted with the following:

- The name of the affected machine, as specified in the `T_MACHINE` class, with all the attributes of that machine
- Some event attributes that identify the event as *machine partitioned*

You can use the EventBroker simply by subscribing to system events. Then, instead of having to query for MIB records, you can be informed automatically when events occur in the MIB by receiving `FML` data buffers representing MIB objects.

How the EventBroker Works

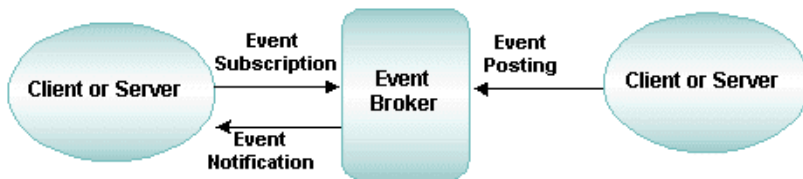
The Oracle Tuxedo EventBroker is a tool through which an arbitrary number of *suppliers of event notifications* can post messages for an arbitrary number of *subscribers*. The suppliers of such notifications may be application or system processes operating as clients or servers. The subscribers of such notifications may be administrators or application processes operating as clients or servers.

Client and server processes using the EventBroker communicate with one another based on a set of *subscriptions*. Each process sends one or more subscription requests to the EventBroker, identifying the event types that the process wants to receive. The EventBroker, in turn, acts like a newspaper delivery person who delivers newspapers only to customers who have paid for a subscription. For these reasons, the paradigm on which the EventBroker is based is described as *publish-and-subscribe* communication.

Event suppliers (either clients or servers) notify the EventBroker of events as they occur. We refer to this type of notification as *posting* an event. Once an event supplier posts an event, the EventBroker matches the posted event with the subscribers that have subscribed for that event type. Subscribers may be administrators or application processes. When the EventBroker finds a match, it takes the action specified for each subscription; subscribers are notified and any other actions specified by subscribers are initiated.

Figure 1-1 shows how the EventBroker handles event subscriptions and postings.

Figure 1-1 Posting and Subscribing to an Event

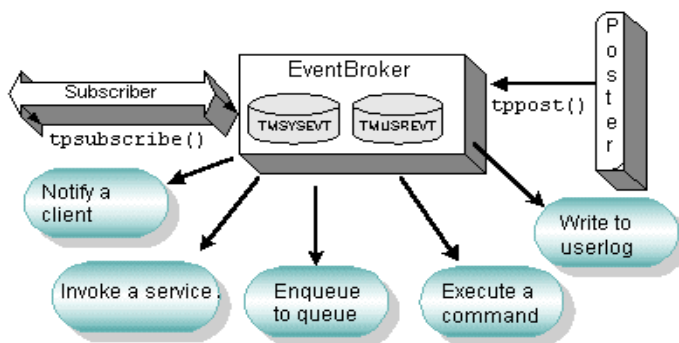


As the administrator for your Oracle Tuxedo application, you can enter subscription requests on behalf of client and server processes through calls to the `T_EVENT_COMMAND` class of the `EVENT_MIB(5)`. You can also invoke the `tpsubscribe(3c)` function to subscribe, programmatically, to an event by using the EventBroker.

Event Notification Methods

The EventBroker subscription specifies one of the notification methods shown in Figure 1-2.

Figure 1-2 Supported Notification Methods



- Notify a client—the EventBroker keeps track of a client’s interest in particular events and notifies the client, without being prompted, when such an event occurs. For this reason, this method is called *unsolicited notification*.
- Invoke a service—if a subscriber wants event notifications to be passed to service calls, the subscriber process should invoke the `tpsubscribe()` function to provide the name of the service to be called.
- Enqueue message to stable-storage queues—for subscriptions with requests to send event notifications to stable-storage queues, the EventBroker will obtain a queue space, queue name, and correlation identifier. A subscriber specifies a queue name when subscribing to an event. The correlation identifier can be used to differentiate among multiple subscriptions for the same event expression and filter rule, that are destined for the same queue.
- Execute a command—when an event is posted, the buffer associated with it is transformed into a system command that is then executed. For example, the buffer may be changed to a system command that sends an e-mail message. This process must be executed through the MIB.
- Write messages to the user log—when events are detected and matched by the EventBroker, the specified messages are written to the user log, or `ULOG`. This process must be executed through the MIB.

Severity Levels of System Events

The EventBroker assigns one of three levels of severity to system events such as server terminations or network failure.

[Table 1-2](#) shows the severity levels of system events.

Table 1-2 Severity Levels of System Events

The level of severity is	When the EventBroker is informed of . . .
. . .	
ERROR	An abnormal occurrence, such as a server being terminated or a network connection being dropped.

Table 1-2 Severity Levels of System Events

The level of severity is . . .	When the EventBroker is informed of . . .
INFO (short for “Information”)	A state change resulting from a process or a change in the configuration.
WARN (short for “Warning”)	The fact that a client has not been allowed to join the application because it failed authentication. A configuration change that threatens the performance of the application has occurred.

What Are the Benefits of Brokered Events?

- Anonymous communication—the Event Broker enables Oracle Tuxedo programs to subscribe to events in which they are interested and it keeps track of all subscriptions. Therefore, a subscriber to one event does not need to know which programs subscribe to the same event, and a poster of an event does not need to know which other programs subscribe to that event. This anonymity allows subscribers to come and go without synchronizing with posters.
- Decoupling of exception conditions—a publish-and-subscribe communication model allows the software detecting an exception condition to be decoupled from the software handling the exception condition.
- Tight integration with the Oracle Tuxedo system—the EventBroker retains functionality such as message buffers, messaging paradigms, distributed transactions, and ACL permission checks for event postings.
- Variety of notification methods—when a client or server subscribes to a system event (such as the termination of a server) or an application event (such as an ATM machine running out of money), it specifies an action that the EventBroker should take when it is notified that the target event has occurred.

If the subscriber is an Oracle Tuxedo client, it can do one of the following at the time it subscribes:

- Request unsolicited notification
- Name a service routine that should be invoked
- Name an application queue in which the EventBroker should store the data for later processing

If the subscriber is an Oracle Tuxedo server, it can do one of the following at the time it subscribes:

- Specify a service request
- Name an application queue in which the EventBroker should store the data

See Also

- [“Subscribing to Events” on page 1-1](#)
- [“Subscribing, Posting, and Unsubscribing to Events with the ATMI and the EVENT_MIB” on page 1-3](#) in *Introducing Oracle Tuxedo ATMI*
- [EVENT_MIB \(5\)](#) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*
- [tpsubscribe\(3c\)](#) in the *Oracle Tuxedo ATMI C Function Reference*
- [tpunsubscribe\(3c\)](#) in the *Oracle Tuxedo ATMI C Function Reference*

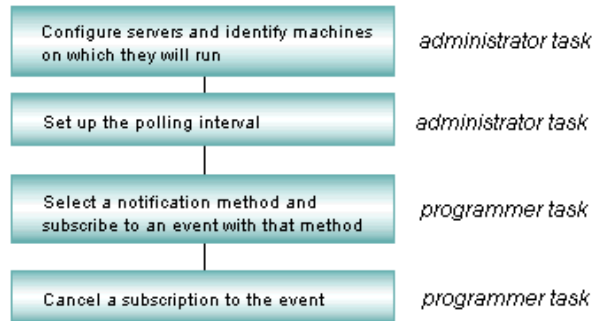
Subscribing to Events

This topic includes the following sections:

- [Process of Using the EventBroker](#)
- [How to Configure EventBroker Servers](#)
- [How to Set the Polling Interval](#)
- [Subscribing, Posting, and Unsubscribing to Events with the ATMI and the EVENT_MIB](#)
- [Subscribing, Posting, and Unsubscribing to Events across Domains](#)
- [How to Select a Notification Method](#)
- [How to Cancel a Subscription to an Event](#)
- [How to Use the EventBroker with Transactions](#)

Process of Using the EventBroker

Use of the EventBroker requires the completion of several preparatory steps. The following flowchart lists these steps and indicates whether each step should be performed by an application administrator or programmer.



For instructions on any of these tasks, click on the appropriate box in the flowchart.

Note: A good way to learn how the EventBroker works is by running *bankapp*, the sample application delivered with the Oracle Tuxedo system. To find out how to copy *bankapp* and run it as a demo, see [“Tutorial for bankapp, a Full C Application”](#) in *Tutorials for Developing Oracle Tuxedo ATMI Applications*.

How to Configure EventBroker Servers

A client accesses the EventBroker through either of two servers provided by the Oracle Tuxedo system: [TMUSREVT \(5\)](#), which handles application events, and [TMSYSEVT \(5\)](#), which handles system events. Both servers process events and trigger the sending of notification to subscribers.

To set up the Oracle Tuxedo EventBroker on your system, you must configure either or both of these servers in the `SERVERS` section of the `UBBCONFIG` file, as shown in the following example.

```
*SERVERS
TMSYSEVT SRVGRP=ADMIN1 SRVID=100 RESTART=Y GRACE=900 MAXGEN=5
CLOPT="-A --"
TMSYSEVT SRVGRP=ADMIN2 SRVID=100 RESTART=Y GRACE=900 MAXGEN=5
CLOPT="-A -- -S -p 90"

TMUSREVT SRVGRP=ADMIN1 SRVID=100 RESTART=Y
MAXGEN=5 GRACE=3600
CLOPT="-A --"
TMUSREVT SRVGRP=ADMIN2 SRVID=100 RESTART=Y
MAXGEN=5 GRACE=3600
CLOPT="-A -- -S -p 120"
```

We recommend that you assign the principal server to the `MASTER` site, even though either server can reside anywhere on your network.

Note: You can reduce the network traffic caused by event postings and notifications by assigning secondary servers to other machines in your network.

How to Set the Polling Interval

Periodically, the secondary server polls the primary server to obtain the current subscription list, which includes filtering and notification rules. By default, polling is done every 30 seconds. If necessary, however, you can specify a different interval.

You can configure the polling interval (represented in seconds) with the `-p` command-line option in `TMUSREVT(5)` or `TMSYSEV(5)` entries in the configuration file, as follows:

```
-p poll_seconds
```

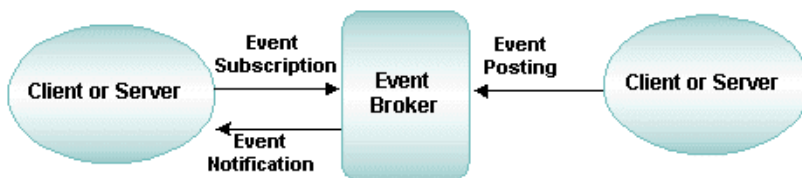
It may appear that event messages are lost while subscriptions are being added and secondary servers are being updated.

Subscribing, Posting, and Unsubscribing to Events with the ATMI and the EVENT_MIB

As the administrator for your Oracle Tuxedo application, you can enter subscription requests on behalf of a client or server process through calls to the `T_EVENT_COMMAND` class of the `EVENT_MIB(5)`. You can also use invoke the `tpsubscribe(3c)` function to subscribe, programmatically, to an event.

[Figure 1-1](#) shows how clients and servers use the EventBroker to subscribe to events, to post events, and to unsubscribe to events.

Figure 1-1 Subscribing to an Event



Identifying Event Categories Using `eventexpr` and `filter`

Clients or servers can subscribe to events by calling `tpsubscribe(3c)`. The `tpsubscribe()` function takes one required argument: *eventexpr*. The value of *eventexpr* can be a wildcard string that identifies the set of event names about which the user wants to be notified. Wildcard strings are described on the `tpsubscribe(3c)` reference page in the *Oracle Tuxedo ATMI C Function Reference*.

As an example, a user on a UNIX system platform who wants to be notified of all events related to the category of networking can specify the following value of *eventexpr*:

```
\.SysNetwork.*
```

The backslash preceding the period (.) indicates that the period is literal. (Without the preceding backslash, the period (.) would match any character except the end-of-line character.) The combination `.*` at the end of `\.SysNetwork.*` matches zero or more occurrences of any character except the end-of-line character.

In addition, clients or servers can filter event data by specifying the optional *filter* argument when calling `tpsubscribe()`. The value of *filter* is a string containing a Boolean filter rule that must be evaluated successfully before the EventBroker posts the event.

As an example, a user who wants to be notified *only* about system events having a severity level of `ERROR` can specify the following value of *filter*:

```
"TA_EVENT_SEVERITY='ERROR' "
```

When an event name is posted that evaluates successfully against *eventexpr*, the EventBroker tests the posted data against the filter rule associated with *eventexpr*. If the data passes the filter rule or if there is no filter rule for the event, the subscriber receives a notification along with any data posted with the event.

Accessing the EventBroker

Your application can access the EventBroker through either the ATMI or the `EVENT_MIB(5)`. [Table 1-1](#) describes both methods.

Table 1-1 Accessing the EventBroker

Method	Function	Purpose
ATMI	<code>tppost(3c)</code>	Notifies the EventBroker, or posts an event and any accompanying data. The event is named by the <i>eventname</i> argument and the <i>data</i> argument, if not NULL, points to the data. The posted event and data are dispatched by the Oracle Tuxedo EventBroker to all subscribers with subscriptions that successfully evaluate against <i>eventname</i> and optional filter rules that successfully evaluate against <i>data</i> .
	<code>tpsubscribe(3c)</code>	Subscribes to an event or a set of events named by <i>eventexpr</i> . Subscriptions are maintained by the Oracle Tuxedo EventBroker, and are used to notify subscribers when events are posted via <code>tppost()</code> . Each subscription specifies one of the following notification methods: client notification, service calls, message enqueueing to stable-storage queues, executing of commands, and writing to the user log. Notification methods are determined by the subscriber's process type (that is, whether the process is a client or a server) and the arguments passed to <code>tpsubscribe()</code> .
	<code>tpunsubscribe(3c)</code>	Removes an event subscription or a set of event subscriptions from the Oracle Tuxedo EventBroker's list of active subscriptions. <i>subscription</i> is an event subscription handle returned by <code>tpsubscribe()</code> . Setting <i>subscription</i> to the wildcard value, -1, directs <code>tpunsubscribe</code> to unsubscribe to all nonpersistent subscriptions previously made by the calling process. Nonpersistent subscriptions are those made without the TPEVPERSIST bit setting in the <code>ctl->flags</code> parameter of <code>tpsubscribe()</code> . Persistent subscriptions can be deleted only by using the handle returned by <code>tpsubscribe()</code> .
EVENT_MIB(5)	N/A	<p>The EVENT_MIB is a management information base (MIB) that stores subscription information and filtering rules. In your own application, you cannot define new events for the Oracle Tuxedo EventBroker using EVENT_MIB, but you can customize the EventBroker to track events and notify subscribers of occurrences of special interest to the application.</p> <p>You can use the EVENT_MIB to subscribe to an event, or to modify or cancel a subscription.</p>

Note: `tpost(3c)`, `tpsubscribe(3c)`, and `tpunsubscribe(3c)` are C functions. Equivalent routines (`TPPOST(3cbl)`, `TPSUBSCRIBE(3cbl)`, and `TPUNSUBSCRIBE(3cbl)`) are provided for COBOL programmers. See the *Oracle Tuxedo ATMI C Function Reference* and the *Oracle Tuxedo ATMI COBOL Function Reference* for details.

Subscribing, Posting, and Unsubscribing to Events Across Domains

Overview

Tuxedo is now equipped to subscribe, post, and unsubscribe brokered events in cross domain environment.

To realize such feature, two new sections, `DM_EVT_IN` and `DM_EVT_OUT`, are added to `DMCONFIG` to manage static event in/out information.

For details of `DM_EVT_IN` and `DM_EVT_OUT`, see “[DMCONFIG\(5\)](#)” in *Tuxedo Reference Guide*.

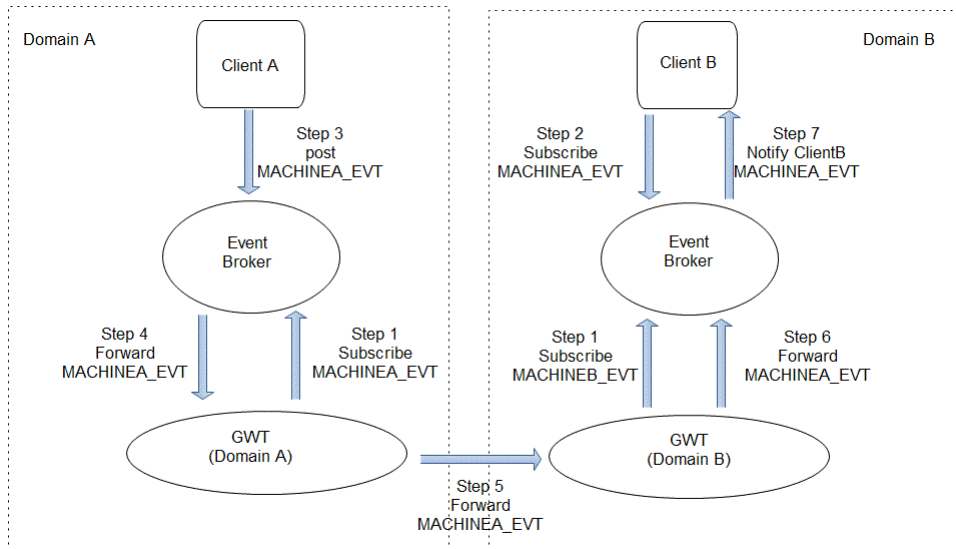
Note: In `UBBCONFIG`, the `EvtBroker` server should be configured prior to the `GWT` server as `GWT` will subscribe the configured events to the `EvtBroker` when starting up.

Configurations in DMLCONF

How to Process Brokered Events Crossing Domains

[Figure 1-2](#) as below illustrates a typical processing flow of subscribing, posting, and unsubscribing a brokered event in cross domain environment.

Figure 1-2 Cross Domain Event Overall Flow



How to Configure DMCONFIG — Case Study

This use case elaborates how to get DMCONFIG well configured.

As shown on [Figure 1-2](#), two clients (Client A and Client B) are located in two domains (Domain A and Domain B), each has one machine within SHM mode (Machine A and Machine B).

For machine A, use `dmloadcf` to create new BDMCONFIG with additional configurations in DMCONFIG as below, and then `tmboot Tuxedo`.

```
*DM_EVT_IN
MACHINEB_EVT

LACCESSPOINT=DOMAINA

*DM_EVT_OUT
MACHINEA_EVT

LACCESSPOINT= DOMAINA
RACCESSPOINT= DOMAINB
```

For machine B, use `dmloadcf` to create new BDMCONFIG with additional configurations in DMCONFIG as below, and then `tmboot Tuxedo`.

```

*DM_EVT_IN
MACHINEA_EVT

LACCESSPOINT=DOMAINB

*DM_EVT_OUT
MACHINEB_EVT

LACCESSPOINT= DOMAINB
RACCESSPOINT= DOMAINA

```

After configuring as above, take a two-step test as below by two clients.

1. Client B issues `tpsubscribe ("MACHINEA_EVT")` on Machine B;
2. Client A issues `tppost ("MACHINEA_EVT")` on Machine A.

Result: Client B will receive the event `MACHINEA_EVT` if `DMCONFIG` is configured correctly.

In cross domain environment, all events should be explicitly imported or exported — requests for an unknown domain will not be accepted. Once configured correctly, GWT server will automatically subscribe every configured event to the local Event Broker when Tuxedo starts up. When receiving a remote event message, local GWT will forward this request to Event Broker. On the other side, when a local event is posted, the Event Broker will forward this event to the local GWT which has subscribed such event. After that, the local GWT will forward this event to the configured remote domain's GWT.

Dynamically Modifying the Event Configurations

Besides allowing users to set up static configurations as above, Tuxedo provides two administration methods to dynamically modify the event configurations as needed without shutting the system down: `dmadmin` command and MIB operations.

For “`dmadmin`” command, two sub-commands (“`advertiseevent`” and “`unadvertiseevent`”) and two sections (“`EVENTS_IN`” and “`EVENTS_OUT`”) are added to support the modification of event configurations dynamically. Related classes are added in MIB operations.

For detailed information, see [dmadmin\(1\)](#) in *Tuxedo Command Reference*, and [DM_MIB\(5\)](#) in *Tuxedo Reference Guide*.

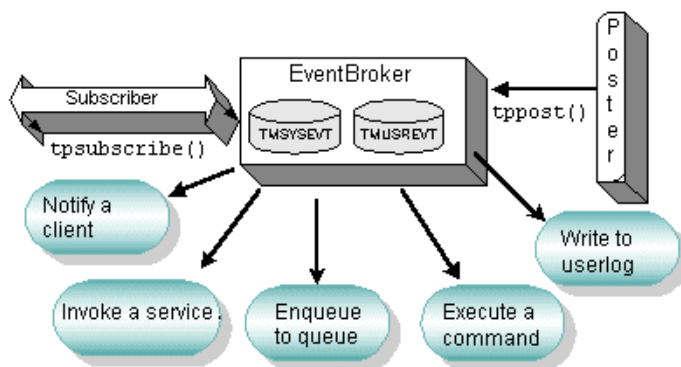
Interoperability

The cross domain event broker feature is supported only when both GWT and EvtBroker are running Oracle Tuxedo 12c Release 1 (12.1.1) or higher.

How to Select a Notification Method

The EventBroker supports a variety of methods for notifying subscribers of events, as shown in Figure 1-3.

Figure 1-3 Notification Methods Supported by the EventBroker



Whichever notification method you choose, the procedure for implementing it is the same: in your call to `tpsubscribe()`, specify an argument that refers to a structure of type `TPEVCTL`.

If the value of the argument is `NULL`, the EventBroker sends an unsolicited message to the subscriber. Two of these methods, having the notification sent to a service and having it sent to a queue in stable storage, cannot be requested directly by a client. Instead, a client must invoke a service routine to subscribe on its behalf.

For each subscription, you can select any of the following notification methods. The EventBroker can:

- Notify the client—the EventBroker keeps track of events in which the client is interested and sends unsolicited notifications to the client when they occur. Some events are anonymously posted. A client can join an application, regardless of whether any other clients have subscribed, and post events to the EventBroker. The EventBroker matches these events against its database of subscriptions and sends an unsolicited notification to the appropriate clients. (See the definition of the `T_EVENT_CLIENT` class in the [EVENT_MIB \(5\)](#) entry in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.)

- Invoke a service—if a subscriber wants event notifications to be sent to service calls, then the `ctl` parameter must point to a valid `TPEVCTL` structure. (See the definition of the `T_EVENT_SERVICE` class in the [EVENT_MIB \(5\)](#) entry in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.)
- Enqueue messages to stable-storage queues—for subscriptions to stable-storage queues, a queue space, queue name, and correlation identifier are specified, in addition to values for `eventexpr` and `filter`, so that matching can be performed. The correlation identifier can be used to differentiate among several subscriptions characterized by the same event expression and filter rule, and destined for the same queue. (See the definition of the `T_EVENT_QUEUE` class in the [EVENT_MIB \(5\)](#) entry in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.)
- Execute commands—using the `T_EVENT_COMMAND` class of the `EVENT_MIB`, subscribers can invoke an executable process. When a match is found, the data is used as the name of the executable process and any required options. (See the definition of the `T_EVENT_COMMAND` class in the [EVENT_MIB \(5\)](#) entry in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.)
- Write messages to the user log (`ULOG`)—using the `T_EVENT_USERLOG` class of the `EVENT_MIB`, subscribers can write system `USERLOG` messages. When events are detected and matched, they are written to the `USERLOG`. (See the definition of the `T_EVENT_USERLOG` class in the [EVENT_MIB \(5\)](#) entry in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.)

How to Cancel a Subscription to an Event

When a client leaves an application by calling `tpterm(3c)`, all of its subscriptions are *canceled* unless the subscription is specified as persistent. (If persistent, the subscription continues to receive postings even after a client performs a `tpterm()`.) If the client later rejoins the application and wants to renew those subscriptions, it must subscribe again.

A well-behaved client unsubscribes before calling `tpterm()`. This is accomplished by issuing a `tpunsubscribe(3c)` call before leaving an application.

How to Use the EventBroker with Transactions

Special handling is needed to use the EventBroker with transactions.

- Before you can use the EventBroker with transactions, you must configure the `NULL_TMS` parameter with the [TMUSREVT \(5\)](#) server for the server groups in which the EventBroker is running.

- The advantage of posting an event in a transaction is that all of the work, including work not related to the posting, is guaranteed to be complete if the transaction is successful. If any work performed within the transaction fails, it is guaranteed that all the work done within the transaction will be rolled back. The disadvantage is that the poster takes a risk that something may cause the transaction to be aborted, and the posting will be lost.
- To specify that a subscription is part of a transaction, use the `TPEVTRAN` flag with `tpssubscribe(3c)`. If the subscription is made transactionally, the action taken in response to an event will be part of the caller's transaction.

Note: This method can be used only for subscriptions that cause an Oracle Tuxedo service to be invoked, or that cause a record to be enqueued on a permanent queue.

How Transactions Work with the EventBroker

If both a poster and a subscriber agree to link their transactions, they create a form of voting. The poster makes an assertion that something is true and infects the message with this transaction. (In other words, the message that leaves the originating process is marked as being associated with the transaction.) The transaction goes to the EventBroker.

The EventBroker's actions, such as calling the service or putting a message in the queue for the subscriber, are also part of the same transaction. If a service routine that is running encounters an error, it can fail the transaction, rolling back everything, including all other transactional subscriptions and the poster's original transaction, which might have invoked other services and performed other database work. The poster makes an assertion ("I'm about to do this"), provides data, and links the data to its transaction.

A number of anonymous subscribers, that is, subscribers about which the poster knows nothing, are invoked transactionally. If any subscriber fails to link its work with the poster's work, the whole transaction is rolled back. All transactional subscribers must agree to link their work with the poster's work, or all the work is rolled back. If a poster has not allowed the posting to participate in its transaction, the EventBroker starts a separate transaction, and gathers all the transactional subscriptions into that transaction. If any of these transactions fail, all the work done on behalf of the transactional subscriptions is rolled back, but the poster's transaction is not rolled back. This process is controlled by the `TPEVTRAN` flag.

Example of Using the EventBroker with Transactions

A stock trade is about to be completed by a brokerage application. A number of database records have been updated by various services during the trade transaction. A posting states that the trade is about to happen.

An application responsible for maintaining an audit trail of such trades has subscribed to this event. Specifically, the application has requested the placement of a record in a specified queue whenever an event of this type is posted. A service routine responsible for determining whether trades can be performed, also subscribes to this type of event; it, too, is notified whenever such a trade is proposed.

If all goes well, the trade is completed and an audit trail is made.

If an error occurs in the queue and no audit trail can be made, the entire stock trade is rolled back. Similarly, if the service routine fails, the transaction is rolled back. If all is successful, the trade is made and the transaction is committed.

See Also

- [?\\$paratext>? on page 1-2](#)
- [“Managing Events Using EventBroker”](#) in *Introducing Oracle Tuxedo ATMI*
- [“Using Event-based Communication”](#) in *Tutorials for Developing Oracle Tuxedo ATMI Applications*
- [tppost\(3c\)](#), [tpsubscribe\(3c\)](#), and [tpunsubscribe\(3c\)](#) in the *Oracle Tuxedo ATMI C Function Reference*
- [TPPOST\(3cb1\)](#), [TPSUBSCRIBE\(3cb1\)](#), and [TPUNSUBSCRIBE\(3cb1\)](#) in the *Oracle Tuxedo ATMI COBOL Function Reference*
- [EVENT_MIB\(5\)](#), [EVENTS\(5\)](#), [TMSYSEVT\(5\)](#), and [TMUSREVT\(5\)](#) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*