webMethods Broker Administration Java API Programmer's Guide

Version 10.15

October 2022

WEBMETHODS

This document applies to webMethods Broker 10.15 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 2007-2022 Software AG, Darmstadt, Germany and/or Software AG USA, Inc., Reston, VA, USA, and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA Inc. and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at https://softwareag.com/licenses/.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at https://softwareag.com/licenses/ and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software AG Products / Copyright and Trademark Notices of Software AG Products". These documents are part of the product documentation, located at https://softwareag.com/licenses/ and/or in the root installation directory of the licensed product(s).

Document ID: PIF-BROKER-JAVA-ADMIN-PROGRAMMERS-GUIDE-1015-20220912

Table of Contents

About this Guide

The *webMethods Broker Administration Java API Programmer's Guide* describes the application programming interfaces (API) that you use to create event-based applications, using the Java language, which perform administrative functions.

This document is intended for use by programmers who are developing event--based applications using webMethods Broker. The reader is assumed to possess general knowledge of programming concepts and specific knowledge of the Java programming language. In addition, you should be familiar with the webMethods Broker systems concepts, administration, and the webMethods Broker APIs.

This book assumes you are familiar with the terminology and basic operations of your operating system. If you are not, see the applicable documentation for that operating system.

Important:
If you have a lower fix level installed, some of the features described in this document might not be available to you. For a cumulative list of fixes and features, see the latest fix readme on the Empower website at https://empower.softwareag.com.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service*, APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at https://documentation.softwareag.com.

In addition, you can also access the cloud product documentation via https://www.softwareag.cloud. Navigate to the desired product and then, depending on your solution, go to "Developer Center", "User Center" or "Documentation".

Product Training

You can find helpful product training material on our Learning Portal at https://knowledge.softwareag.com.

Tech Community

You can collaborate with Software AG experts on our Tech Community website at https://techcommunity.softwareag.com. From here you can, for example:

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.
- Get the latest Software AG news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at https://github.com/softwareag and https://hub.docker.com/publishers/softwareag and discover additional Software AG resources.

Product Support

Support for Software AG products is provided to licensed customers via our Empower Portal at https://empower.softwareag.com. Many services on this portal require that you have an account. If you do not yet have one, you can request it at https://empower.softwareag.com/register. Once you have an account, you can, for example:

- Download products, updates and fixes.
- Search the Knowledge Center for technical information and tips.
- Subscribe to early warnings and critical alerts.
- Open and update support incidents.
- Add product feature requests.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 Getting Started

Overview

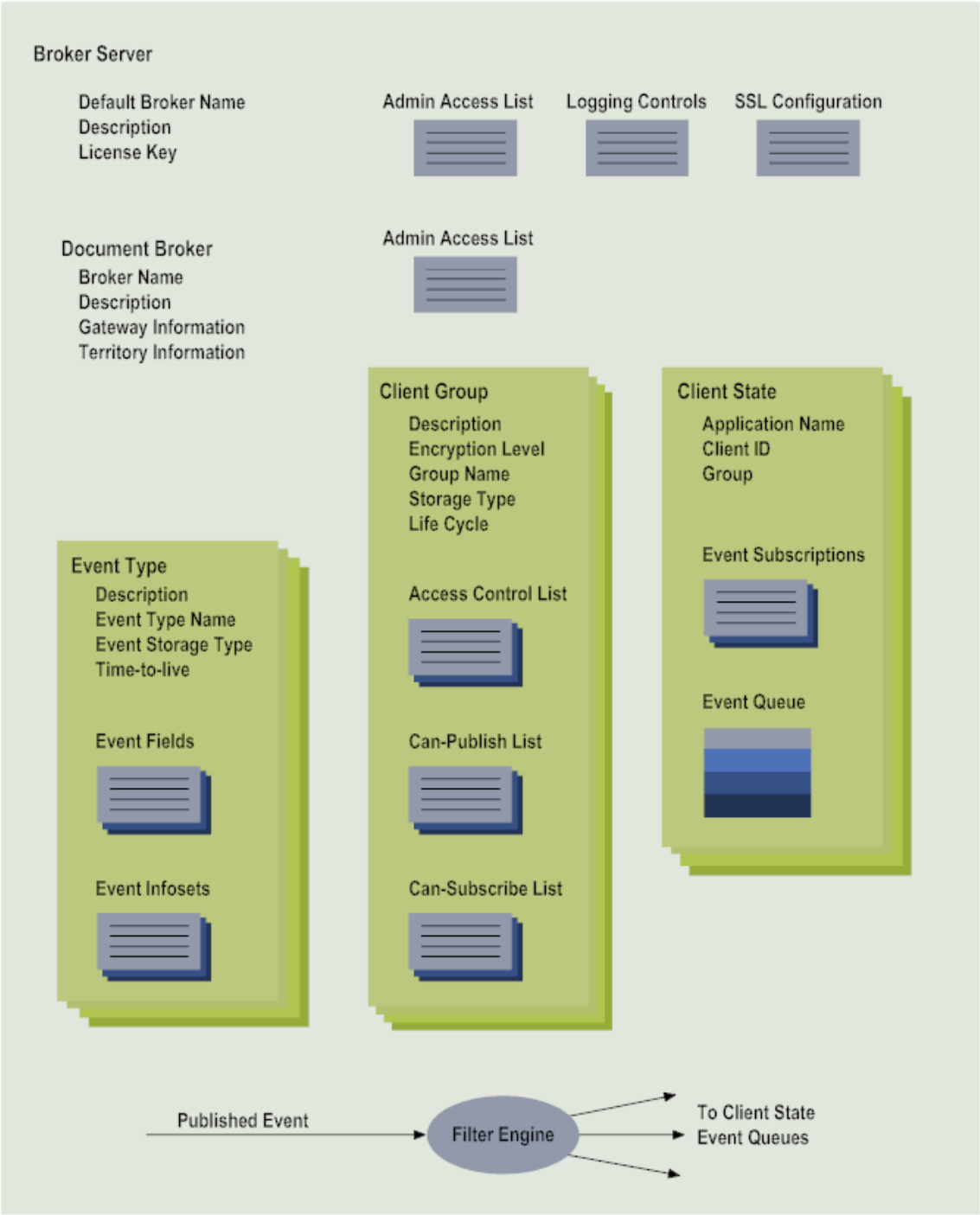
This chapter describes the basic features of the administration API for the webMethods Broker system. The API lets you create applications that automate the monitoring and management of your information Brokers and Broker clients. You can also create applications that integrate your webMethods Brokers and Broker clients with existing network management systems.

After reading this chapter, you will be ready to learn how to use the webMethods Broker administration interfaces in your client applications.

Overview of the webMethods Broker Administration API

The webMethods Broker administration API allows your client applications to control and monitor the basic elements of the webMethods Broker system, including Broker Servers and the hosts on which they are running, Broker clients, client groups, event type definitions, security configurations, and tracing and logging functions. The following figure shows the elements that comprise a webMethods Broker system and their relationship to one another. For a complete list of attributes of Brokers, clients, client groups, and event types, see the *webMethods Broker Client Java API Programmer's Guide* and *Administering webMethods Broker*.

Elements of a webMethods Broker system



BrokerServerClient

The BrokerServerClient class, described in "Managing Brokers and Broker Servers" on page 17, allows your client application to administer the Broker Server on which one or more Brokers are running. The administration methods offered by this interface allow your client application to:

- Start and stop a Broker Server.

- Create and destroy a Broker.
- List all of the Brokers running within a particular Broker Server.
- Obtain statistics for the host where a Broker Server is running.
- Pause all publishing clients on a Broker Server.
- Manage Broker Server security.

"Managing Brokers and Broker Servers" on page 17 describes the use of this class to administer hosts where Brokers are running.

BrokerAdminClient

The BrokerAdminClient interface allows your client application to administer a Broker. The methods offered by this interface allow your client application to:

- List information for all of the Broker clients connected to the Broker.
- List all of the client groups and event types defined for the Broker.
- Create, modify, and destroy client groups and event type definitions.
- Obtain statistics for a Broker client, client group, event type, or a Broker.
- Create and modify territories and clusters.

"Managing Broker Clients" on page 25 and "Managing Client Groups" on page 39 describe the use of this interface to administer Broker clients and client groups. "Managing Broker Event Types" on page 47 covers the methods offered for managing event types.

Monitoring Brokers

The webMethods Broker system provides a variety of ways for your client applications to monitor a Broker's activity. "Monitoring Broker Activity" on page 93 describes the interfaces and events available to your client application for monitoring Brokers.

Managing Security Configurations

TheBrokerAccessControlList, BrokerServerClient, BrokerAdminClient, and BrokerSSLConfigV2 interfaces allow your client application to control the security configuration for Broker Servers and Brokers. "Managing Security Configurations" on page 55covers the interfaces offered for managing security configurations.

Managing Territories

webMethods Broker allows two or more Brokers to cooperate as a territory. The use of territories allows an event published by a client connected to one Broker to be forwarded to a client connected to a different Broker within the same territory.

You may also configure two Brokers so that they form a gateway between two different territories.

"Managing Broker Territories" on page 63 describes the interfaces you can use to configure and manage Broker territories.

Managing Clusters

webMethods Broker allows you to group two or more Brokers to form a cluster. The Brokers in a cluster share information about their event type definitions and client groups. The clusters can be configured with JMS load balancing policies for enabling client-side load balancing and failover. For more information, see the *webMethods Broker Messaging Programmer's Guide*.

You may also configure two Brokers so that they form a gateway between two different clusters.

"Managing Broker Clusters" on page 75 describes the interfaces you can use to configure and manage Broker clusters.

Managing Configurations

The webMethods Broker API for Java provides interfaces that allow you to load, store, import and export configuration information for Broker Servers, Brokers, Broker clients, client groups, and event types. These interfaces are described in "Managing Site Configurations" on page 85.

Using the Administration API for Java

The webMethods Broker administration API is implemented as a Java package that provides all the necessary interfaces, classes and methods for creating administrative client applications.

Compatibility

This version of the webMethods Broker Java API, and applications developed with it, are compatible and fully inter-operable with Brokers and applications developed with the C, Java, and ActiveX APIs and webMethods 9.x and later.

Application Development with webMethods Broker

For information on application development, see the *webMethods Broker Client Java API Programmer's Guide*.

Application Deployment

When deploying your administrative client applications, you must also deploy the necessary webMethods Broker classes. The webMethods installation contains an archive with all of the files necessary for application deployment.

Note:
These files must be in the application's `CLASSPATH` when the application is deployed.

The default location of the files required for application deployment are as follows:

Platform	Default location
AIX and Linux	■ *Software AG_directory* /common/lib/wm-brokerclient.jar ■ *Software AG_directory* /common/lib/wm-g11nutils.jar
Windows	■ *Software AG_directory* \common\lib\wm-brokerclient.jar ■ *Software AG_directory* \common\lib\wm-g11nutils.jar

webMethods Broker Java API Documentation

The webMethods Broker Java API provides API-level documentation. The documentation is available as HTML web pages, in the following platform-specific location.

The default location of the webMethods Broker Java API documentation is as follows:

Platform	Location of webMethods Broker Administration Java API Packages	Location of webMethods Broker Java Client API Packages
AIX and Linux	*Software AG_directory* /Broker/doc/java_api/index.html	*Software AG_directory* /Broker/doc/java_api/index.html
Windows	*Software AG_directory* \Broker\doc\java_api\index.html	*Software AG_directory* \Broker\doc\java_api\index.html

2 Managing Brokers and Broker Servers

Overview

This chapter describes how you can use the BrokerServerClient interface to manage the Messaging Brokers running in a particular Broker Server. Reading this chapter will help you understand:

- Using a BrokerServerClient to list all the Brokers, including the default Broker, running in a Broker Server.
- Using a BrokerServerClient to create and destroy a Broker.
- How to create and destroy a BrokerServerClient object.
- How to register callback methods for connection notification.
- Using a BrokerServerClient to obtain usage statistics for a Broker Server.
- Using a BrokerServerClient to get and set Broker Server properties, such as the default Broker name and the license string.

Understanding Broker Server Clients

A client program creates a BrokerServerClient object to manage a Broker Server's configuration and to control or monitor a Broker in that Broker Server.

For example, a network monitoring application might create a BrokerServerClient to learn how much swap space is available or the number of Brokers in a Broker Server. A network management application might use a BrokerServerClient to create a Broker on a particular host at 7AM in the morning and destroy it at 7PM at night.

Creating a BrokerServerClient

Creating a BrokerServerClient will establish a connection between your application and a Broker Server process on a particular host. Your application must specify the name of the host when creating the BrokerServerClient object.

The following example illustrates how to create a BrokerServerClient:

```
import COM.activesw.api.client.*;
class myMonitor
{
   static String broker_host = "localhost";
   .  .  .
   public static void main(String args[])
   {
       BrokerServerClient c;
       .  .  .
       /* Create a Broker Server client */
       try {
       c = new BrokerServerClient(broker_host, null);
       } catch (BrokerException ex) {
        System.out.println("Error on create Broker Server client\n"+ex);
          return;
      }
```

```
.  .  .
```

Destroying a BrokerServerClient

The following example illustrates the use of the BrokerServerClient.destroy method, which disconnects the client from the host.

```
.  .  .
BrokerServerClient c;
.  .  .
try {
 c.destroy();
} catch (BrokerException ex) {
 System.out.println("Error on Broker Server client destroy\n"+ex);
 return;
}
.  .  .
```

Connection Notification

The connection notification feature allows you to register a callback method for a particular Broker Server client that will be invoked if the client is disconnected from the server.

Note:
Connection callback methods do not have a global scope. They must be registered separately for each server client that wants to use this feature.

Defining a Callback Object

You use BrokerServerConnectionCallback to derive your own callback object. You must provide an implementation of the handleHostConnectionChange method to perform the processing you desire. You can implement this method to recreate any needed client state that may have been lost or to provide other functions, such as logging of error messages.

Registering the Callback Object

You use the BrokerServerClient.registerConnectionCallback method to register a method you want to be called in the event your Broker client is disconnected or reconnected to its Broker. This method accepts two parameters. The first parameter is the BrokerConnectionCallback-derived object that implements your callback method. The second parameter is a client_data object, which is used to pass any needed data to the callback method.

Note:
Any callback objects previously registered for a Broker client will be replaced by the one currently being registered.

When the BrokerServerConnectionCallback object's the callback method is invoked, that method's connect_state parameter will be set to one of the following BrokerClient-defined values shown in the following table.

connect_state	Meaning
CONNECT_STATE_CONNECTED	The Client connection has been re-established, because automatic reconnect was enabled.
CONNECT_STATE_DISCONNECTED	The client has been disconnected.
CONNECT_STATE_RECONNECTED	The Client was disconnected, but the connection was re-established immediately. This only happens if the automatic reconnect feature is enabled and the connection is re-established before a disconnected state can be reported.

Un-Registering Callback Objects

You can un-register a callback by invoking the BrokerServerClient.registerConnectionCallback method with a null callback object.

Managing a Broker Server Configuration

Getting Broker Server Statistics

You can use the BrokerServerClient.getStats method to obtain statistics for the Broker Server to which your BrokerServerClient is connected. The statistics are returned as fields within a BrokerEvent.

After you have obtained the BrokerEvent containing the statistics, you can retrieve each field using the appropriate BrokerEvent.get<type>Field method. The following example illustrates how to obtain and parse the BrokerEvent containing the Broker Server's statistics.

```
.  .  .
int num_connections;
boolean low_disk_space;
BrokerEvent e;
BrokerServerClient c;
try {
 /* Create a Broker Server client */
 c = new BrokerServerClient(broker_host, null);

 /* Get the BrokerEvent containing the Broker Server’s statistics */
 e = c.getStats();
 /* Get the total number of connections */
 num_connections = e.getIntegerField( "numConnections");
 /* Get the disk space status */
 low_disk_space = e.getBooleanField( "isDiskSpaceLow");
 .  .  .
} catch (BrokerException ex) {
 System.out.println("Error getting host statistics\n"+ex);
 return;
}
.  .  .
```

Getting System Usage Statistics

The BrokerServerClient.getUsageStats method allows you to obtain system usage statistics for the host to which your BrokerServerClient is connected.

After you have obtained the BrokerEvent containing the statistics, you can retrieve each field using the BrokerEvent.get<type>Field method. The version of the get<type>Field method to be used will depend on the data type of the field you are retrieving. The following example illustrates how to obtain a BrokerEvent containing the system usage statistics and extract the value of two fields.

```
.  .  .
long swap_space_total;
long swap_space_free;
BrokerEvent e;
BrokerServerClient c;
try {
 /* Create a Broker Server client */
 c = new BrokerServerClient(broker_host, null);

 /* Get the BrokerEvent containing the usage statistics */
 e = c.getUsageStats();
 /* Get the total amount of swap space */
 swap_space_total = e.getLongField( "swapSpaceMax");
 /* Get the swap space available */
 swap_space_free = e.getLongField( "swapSpaceFree");
 .  .  .
} catch (BrokerException ex) {
 System.out.println("Error getting usage statistics\n"+ex);
 return;
}
.  .  .
```

Getting and Setting the Default Broker

You can use the BrokerServerClient.getDefaultBrokerName method to obtain the name of the default Broker on the Broker Server to which your BrokerServerClient is connected. If you do not specify a Broker name when you create a BrokerClient, your client will be connected to the default Broker.

You can use the BrokerServerClient.setDefaultBrokerName method to set the name of the default Broker.

The following example illustrates how to use the getDefaultBrokerName and setDefaultBrokerName methods to get and set the default Broker name.

```
.  .  .
String broker_name;
BrokerServerClient c;
.  .  .
 /* Create a Broker Server client */
 c = new BrokerServerClient(broker_host, null);
 .  .  .
try {
 /* Get the default Broker name */
 broker_name = c.getDefaultBrokerName();
} catch (BrokerException ex) {
 System.out.println("Error getting default Broker\n"+ex);
```

```
 return;
}
try {
 /* Set the default Broker name */
 c.setDefaultBrokerName("My Broker");
 .  .  .
} catch (BrokerException ex) {
 System.out.println("Error setting default Broker\n"+ex);
 return;
}
.  .  .
```

Getting and Setting Log Configuration

You can use the BrokerServerClient.getLogConfig method to obtain the logging configuration for a Broker Server and the BrokerLogConfigclass represents logging configuration.

The BrokerServerClient.setLogConfig method allows you to set the logging configuration for a Broker Server.

Getting the Port Information

The BrokerServerClient.getActivePort method allows you to obtain the base port for the Broker Server to which your BrokerServerClient is connected. The base port is used for non-SSL connections and has a default value of 6849. The base port number is also used to calculate the port numbers as shown in the following table.

Port number	Description
Base port number minus 1	Used for SSL connections without client authentication.
Base port number minus 2	Used for SSL connections with client authentication.

Getting and Setting the License

The BrokerServerClient.getLicense, and BrokerServerClient.getLicenseContent methods let you obtain the license details for the Broker Server to which your BrokerServerClient is connected.

The BrokerServerClient.setLicense, BrokerServerClient.setLicenseContent, and BrokerServerClient.setLicenseFile methods are used to set the license.

Note:
For Broker Servers earlier than 8.0 version, the BrokerServerClient.getLicense method returns a string containing the Broker host's license.

Starting and Stopping the Host Process

You can use the BrokerServerClient.startServerProcess method to start the Broker Server process on the specified host. The stopProcess method is used to stop a Broker Server process.

The getServerProcessRunStatus method can be used to determine the state of the Broker Server process.

Managing Security Configurations

This topic is covered in "Managing Security Configurations" on page 55.

Managing Brokers

The BrokerServerClient class provides several methods for monitoring and controlling Brokers. The BrokerAdminClient also offers a limited number of methods, described in " Broker Administration" on page 30, for monitoring a particular Broker.

Using Change Locks

If you have multiple client applications that want to manage a Broker, race conditions may result if more than one client attempts to update the Broker's configuration.

You can use the BrokerAdminClient.acquireChangeLock method to obtain a lock for the Broker to which a BrokerClient is connected. This method will return a BrokerChangeLock object with the variables shown in the following table.

Data Member	Description
`BrokerBoolean acquired;`	Set to 1 (true) if the lock was acquired by this BrokerClient, otherwise set to 0 (false).
`String client_id;`	If `acquired` is 0 (false), this member will contain the ID of the client currently holding a lock on the Broker. Otherwise, this member will be set to `NULL`. For information on client IDs, see " Broker Client Administration" on page 31.
`int session_id;`	If `acquired` is 0 (false), this member will contain the session ID of the client currently holding a lock on the Broker. Otherwise, this member will be set to zero.
`BrokerDate lock_time`	If `acquired` is 0 (false), this member will contain the date and time the lock was acquired. Otherwise, this member will be set to `NULL`.

Your client application should use the BrokerAdminClient.releaseChangeLock method to release a Broker change lock, after it has finished updating the Broker.

Listing all Brokers on a Host

The BrokerServerClient.getBrokers method lets you obtain a list of all the Brokers running in the Broker Server to which your BrokerServerClient is connected. An array of BrokerInfo objects is returned and each element contains details such as a Broker's name, description, and territory information. This method is available to all Broker Server clients.

The following example illustrates how to list all Brokers in a Broker Server:

```
.  .  .
int i;
BrokerInfo brokers[];
BrokerServerClient c;
try {
 /* Create a Broker Server client */
 c = new BrokerServerClient(broker_host, null);
 /* Get a list of all Brokers in this server and print them out*/
 brokers = c.getBrokers();
 for(i = 0;  i < brokers.length;  i++) {
  System.out.println(brokers[i].broker_name+"\n");
 }
} catch (BrokerException ex) {
 System.out.println("Error getting Broker list\n"+ex);
 return;
}
.  .  .
```

Creating and Destroying Brokers

You can use the BrokerServerClient.createBrokermethod to create a Broker in the Broker Server to which your BrokerServerClient is connected. You must supply the Broker's name and description. You may also specify that the Broker you are creating should be the default Broker. Your client must have administrative permission to with Broker Server.

The following example illustrates how to create a Broker:

```
.  .  .
BrokerServerClient c;
try {
 /* Create a Broker Server client */
 c = new BrokerServerClient(broker_host, null);

 /* Create a Broker and make it the default */
 c.createBroker( "My Broker", "This is my Broker", true);
 ...
} catch (BrokerException ex) {
 System.out.println("Error creating Broker\n"+ex);
 return;
}
.  .  .
```

You can use the BrokerServerClient.destroy method to destroy a BrokerServerClient object. The BrokerServerClient will be disconnected from the Broker Server.

3 Managing Broker Clients

Overview

This chapter describes how you can use the BrokerAdminClient interface to obtain information about Broker clients that are connected to a Broker. Reading this chapter should help you understand:

- How to create and use a BrokerClient for administration purposes.
- How to disconnect and reconnect an administrative BrokerClient.
- How to obtain the identifiers of the Broker clients connected to a particular Broker.
- How to obtain statistics for Broker clients connected to a Broker.
- Getting or setting a Broker client's properties, including event subscriptions.
- Destroying or disconnecting a Broker client.

Understanding BrokerAdminClients

The BrokerAdminClient class extends the BrokerClient class. In addition to having access to all the BrokerClient methods, the BrokerAdminClient provides a wide array of methods for administering Information Brokers, Broker clients, client groups, and event type definitions. Your client program creates a BrokerAdminClient to monitor and control the operation of a particular Broker.

For example, a network monitoring application might create a BrokerAdminClient to:

- Obtain usage statistics for a Broker, a particular Broker client, a client group, or an event type.
- Obtain a list of a Broker's clients, client groups, and event type definitions.
- Create or destroy a Broker client, client group, or event type definition.
- Change the properties of a Broker client, client group, or event type definition.

This chapter focuses on administration methods related to Brokers and Broker clients. "Managing Client Groups" on page 39 describes the BrokerAdminClient methods for administering client groups and "Managing Broker Event Types" on page 47 describes the methods for administering event type definitions.

Creating a BrokerAdminClient

Creating a BrokerAdminClient will establish a connection between your application and a Broker. This connection is identified by the following *tuple* (an ordered set of values):

- The name of the host where the Broker server is running.
- The IP port number assigned to the Broker's server.
- The name of the Broker.

Multiple Broker clients within a single application that connect to the same Broker will all share a single network connection.

Your application creates an administrative Broker client by using the BrokerAdminClient constructor and specifying these parameters:

- The name of the host where the Broker to which you want to connect is running.
- The name of the Broker to which you wish to connect. You may specify a `null` value if you want to connect to the default Broker. The default Broker for a particular host is determined by your webMethods Broker administrator. It can also be set or obtained using the BrokerServerClient class, described in "Managing Brokers and Broker Servers" on page 17. See "Listing all Brokers on a Host" on page 24 for details on obtaining the names of Brokers available on a given host.
- A unique client ID that identifies your Broker client. You may specify a `null` value if you wish the Broker to generate a unique client ID for you. The client ID generated by the Broker can be obtained after this call completes by calling the BrokerClient.getClientId method.
- The client group for your new Broker client. Client groups define the event types your new Broker client will be able to publish or retrieve, as well as the life cycle and queue storage type for your Broker client. Client groups are defined by your webMethods Broker administrator. See "Managing Client Groups" on page 39 for information on using the BrokerAdminClient class to administer client groups.
- The name of the application that is creating the Broker client. This name is used primarily by webMethods Broker administration and analysis tools. The application name can be any meaningful string of characters you choose.
- A BrokerConnectionDescriptor to be used for the new Broker client. If you specify a `null` value, a new connection descriptor will be created for you.

The following example illustrates how to create a new BrokerAdminClient object. In this example, a `null` Broker name is passed to indicate that the caller wishes to connect to the default Broker on the host "localhost". A `null` client ID is specified, indicating that the Broker should generate a unique client ID. A client group named "admin" is requested and the application name is set to "My Broker Monitor." A `null` connection descriptor is passed, indicating that the caller wishes a default connection to be established.

```
import COM.activesw.api.client.*;

class MyMonitor
{
 static String broker_host = "localhost";
 static String broker_name = null;
 static String client_group = "admin";
 .  .  .
 public static void main(String args[])
 {
  BrokerAdminClient c;
  .  .  .
  /* Create an administrative client */
  try {
   c = new BrokerAdminClient(broker_host, broker_name, null,
     client_group, "My Broker Monitor", null);
  } catch (BrokerException ex) {
   System.out.println("Error creating admin client\n"+ex);
     return;
```

```
 }
.  .  .
```

Destroying a BrokerAdminClient

The following example shows the use of the BrokerAdminClient.destroy method, which is inherited from the BrokerClient class. The client state and event queue for the BrokerAdminClient that is being destroyed will also be destroyed.

```
.  .  .
BrokerAdminClient c;
.  .  .
try {
 c.destroy();
} catch (BrokerException ex) {
 System.out.println("Error while destroying admin client\n"+ex);
 return;
}
.  .  .
```

Using Multiple BrokerAdminClients

You may find that a number of programming situations are made easier by creating multiple BrokerAdminClient objects within a single application.

- If your application needs to monitor several Brokers at once, you can use a separate client for each Broker.
- Because different Broker clients can belong to distinct client groups, they can subscribe to and publish different event types and their administrative permissions may be different.
- A multi-threaded application can spawn separate threads for each Broker client, which can result in a cleaner programming model.

The webMethods Broker API allows you to disconnect a Broker client from a Broker without actually destroying the Broker client. This is only useful if your Broker client's life cycle is *explicit-destroy* because the client state for the Broker client and all queued events will have been preserved by the Broker. Any events received by the Broker for your Broker client while it was disconnected will remain in the Broker client's event queue. When your Broker client reconnects to the Broker, specifying the same client ID, it can continue retrieving events from the event queue. Disconnecting and reconnecting can be particularly useful for applications that wish to do batch-style processing at scheduled intervals.

Disconnecting a BrokerAdminClient

You can disconnect your BrokerAdminClient by calling the disconnect method inherited from the BrokerClient class. If the Broker client's life cycle is *destroy-on-disconnect*, the client state and event queue storage will be destroyed by the Broker. If the Broker client's life cycle is explicit-destroy, the client state and event queue storage will be preserved by the Broker until the Broker client reconnects.

The following example illustrates how to disconnect a BrokerAdminClient:

```
.  .  .
BrokerAdminClient c;
.  .  .
try {
 c.disconnect();
} catch (BrokerException ex) {
 System.out.println("Error while disconnecting the admin client\n"+ex);
 return;
}
.  .  .
```

Reconnecting a BrokerAdminClient

You can reconnect a previously disconnected Broker client that has a life cycle of explicit-destroy by invoking the reconnectAdmin method and specifying the same client ID that was used when your client was last connected, as shown in the example below.

If the Broker client's life cycle is explicit-destroy, the client state and event queue storage will have been preserved by the Broker since previous invocation of the disconnect method.

The following example illustrates how to reconnect a BrokerAdminClient:

```
static String broker_host = "localhost";
static String broker_name = null;
static String client_group = "myExplicitDestroyClientGroup";
.  .  .
public static void main(String args[])
{
 BrokerAdminClient c;
 String client_id;
 .  .  .
 /* Create an admin client */
 try {
  c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 } catch (BrokerException ex) {
  .  .  .
 }
 /* Save the client ID */
 client_id = c.getClientId();
 .  .  .
 /* Disconnect the admin client */
 try {
  c.disconnect();
 } catch (BrokerException ex) {
  .  .  .
 }
 .  .  .
  /* Reconnect the admin client with the same client_id */
 try {
  c.reconnectAdmin(broker_host, broker_name,    client_id, null);
 } catch (BrokerException ex) {
  .  .  .
 }
 .  .  .
```

Using the newOrReconnectAdmin Method

You might find it convenient to use the newOrReconnectAdmin method to create or reconnect an administrative client when your client application is expected to be executed repeatedly. The BrokerAdminClient. newOrReconnectAdmin method will attempt to create a new BrokerAdminClient. If the client already exists and was simply disconnected, it will be reconnected.

Broker Administration

The BrokerAdminClient provides a small set of methods for administering a particular instance of a Broker. See "Managing Brokers" on page 23 for information on the administration methods offered by the BrokerServerClient class.

Destroying a Broker

You can use the BrokerAdminClient.destroyBroker method to destroy the Broker to which your BrokerAdminClient is connected. Your client must have permission to destroy the Broker. Destroying the Broker will also destroy your BrokerAdminClient.

The following example illustrates how to destroy a Broker:

```
.  .  .
BrokerAdminClient c;
try {
 /* Create a Broker Admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    "admin", "My_Broker_Monitor");

 /* Destroy a Broker */
 c.destroyBroker( "My_Broker");
 ...
} catch (BrokerException ex) {
 System.out.println("Error destroying Broker\n"+ex);
 return;
}
.  .  .
```

Getting and Setting the Broker Description

You can obtain or set a Broker's description using the BrokerAdminClient.getBrokerDescription and BrokerAdminClient.setBrokerDescription methods.

Getting Broker Statistics

The BrokerAdminClient.getBrokerStats method allows you to obtain statistics for a Broker. The statistics are returned as fields within a BrokerEvent.

After you have obtained the BrokerEvent containing the statistics, you can retrieve each field using the BrokerEvent.get<type>Field method. The version of the get<type>Field method to be used will depend

on the data type of the field you are retrieving. The following example illustrates how to obtain a BrokerEvent containing the system usage statistics and extract the value of two fields.

```
.  .  .
int num_clients;
int num_event_types;
BrokerDate create_date;
BrokerEvent e;
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
   client_group, "My Broker Monitor",null);

 /* Get the BrokerEvent containing the Broker's statistics */
 e = c.getBrokerStats();
 /* Get the time and date that the Broker was created */
 create_date = e.getDateField( "createTime");
 /* Get the number of connected clients */
 num_clients = e.getIntegerField( "numClients");
 /* Get the number of event type definitions */
 num_event_types = e.getIntegerField( "numEventTypes");
 .  .  .
} catch (BrokerException ex) {
 System.out.println("Error getting Broker statistics\n"+ex);
 return;
}
.  .  .
```

Broker Client Administration

Obtaining Broker Client Identifiers

The BrokerAdminClient.getClientIds method lets you obtain a list of all client identifiers known to the Broker to which your BrokerAdminClient is connected and to which your client has access.

The following example illustrates how to list all client identifiers for a Broker:

```
.  .  .
int i;
String client_ids[];
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* Get a list of all client ids for this Broker and print them out*/
 client_ids = c.getClientIds();
 for(i = 0;  i < client_ids.length;  i++) {
  System.out.println(client_ids[i]);
 }
} catch (BrokerException ex) {
 System.out.println("Error getting Client Id list\n"+ex);
 return;
}
.  .  .
```

The BrokerAdminClient.getClientIdsByClientGroup method lets you obtain a list of all client identifiers belonging to the client group which you specify and to which your client has access. The following example illustrates how to list all client identifiers for a client group:

```
.  .  .
int i;
String client_ids[];
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* List of all client ids for this client group and print them out*/
 client_ids = c.getClientIdsByClientGroup("myGroup");
 for(i = 0;  i < client_ids.length;  i++) {
  System.out.println(client_ids[i]);
 }
} catch (BrokerException ex) {
 System.out.println("Error while getting the Client ID list\n"+ex);
 return;
}
.  .  .
```

The BrokerAdminClient.getClientIdsWhichAreSubscribed method lets you obtain a list of all the client identifiers which have subscribed to a particular event type and to which your client has access. The following example illustrates how to list all client identifiers subscribed to an event type:

```
.  .  .
int i;
String client_ids[];
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* Get a list of all the client IDs that are subscribed and print them out*/
 client_ids = c.getClientIdsWhichAreSubscribed("myEvent");
 for(i = 0;  i < client_ids.length;  i++) {
  System.out.println(client_ids[i]);
 }
} catch (BrokerException ex) {
 System.out.println("Error while getting the Client ID list\n"+ex);
 return;
}
.  .  .
```

Getting Broker Client Information

You can use the BrokerAdminClient.getClientInfoById method to obtain details about a particular Broker client, such as its state sharing limit and client group name. The information is returned as a BrokerClientInfo object, which contains the following Broker client information:

- Application name
- State sharing status and limit
- Client group affiliation

- Highest publish sequence number
- All sessions for the Broker client

A Broker client may share its client state, which means that several client applications can create BrokerClient objects using the same client identifier. All client application sessions using a particular client identifier will be described as an array of BrokerClientSession objects.

You can use the BrokerAdminClient.getClientInfosById method to obtain information about all Broker clients for a particular Broker.

The following example illustrates how to obtain the BrokerClientInfo object and extract the value of three fields. In this example, the client identifier is hard-coded. See "Obtaining Broker Client Identifiers" on page 31 for information on discovering client identifiers at run time.

```
.  .  .
BrokerClientInfo inf;
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
   client_group, "My Broker Monitor",null);
 /* Get the information for a particular Broker client */
 inf = c.getClientInfoById("A123");
 System.out.println("Application name: "+inf.app_name);
 System.out.println("Can share state: "+inf.can_share_state);
 System.out.println("Client group: "+inf.client_group);
 .  .  .
} catch (BrokerException ex) {
 System.out.println("Error BrokerClientInfo\n"+ex);
 return;
}
.  .  .
```

Creating Broker Clients

You can use the create BrokerAdminClient.createClient method to create a Broker client on the Broker to which your BrokerAdminClient is connected. You must specify the following information:

- A client identifier
- A client group
- An application name

You can also specify an optional connection description for the Broker client.

Because the resulting Broker client will have no connection to a client application, the life cycle of its client group must be explicit-destroy.

Reconnecting a Remote Broker

You can use the BrokerAdminClient.checkAndRestartForwarding method to restart remote Broker connections, where remote Brokers are territory and gateway Brokers. You can also specify to restart a connection to a specific remote Broker.

Disconnecting Broker Clients

You can use the BrokerAdminClient.disconnectClientById method to disconnect all Broker clients using a specific client identifier. If the Broker client's life cycle is destroy-on-disconnect, the client state and event queue storage will be destroyed by the Broker. If the Broker client's life cycle is explicit-destroy, the client state and event queue storage will be preserved by the Broker until the Broker client reconnects.

Note:
If more than one client application is sharing the same client state, they are using the same client ID and they will all be disconnected.

The following example illustrates how to disconnect a BrokerClient by client identifier. In this example, the client identifier is hard-coded. See "Obtaining Broker Client Identifiers" on page 31 for information on discovering client identifiers at run time.

```
.  .  .
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
   client_group, "My Broker Monitor",null);
 /* Disconnect all Broker clients with a specific client identifier */
 c.disconnectClientById("A123");
} catch (BrokerException ex) {
 System.out.println("Error while disconnecting BrokerClient\n"+ex);
 return;
}
.  .  .
```

Disconnecting by Session Identifier

A Broker client can share its client state, which means that several client applications can create BrokerClient objects using the same client identifier. In these cases, each client application's connection is represented by a BrokerClientSession object with a unique session identifier.

The following example shows how you can disconnect a single Broker client session by invoking the BrokerAdminClient.disconnectClientSessionById method.

```
.  .  .
BrokerAdminClient c;
BrokerClientInfo inf;
int i;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
```

```
    client_group, "My Broker Monitor",null);
  /* Get the information for a particular Broker client */
  inf = c.getClientInfoById("A123");
  /* Get the session ID of the client with an IP
   * address of 101.111.12.1
   */
  for( i = 0;  i < inf.sessions.length;  i++) {
   if( inf.sessions[i].ip_address == 0x656f0c01) {
    /* Disconnect the Broker client by session identifier */
    c.disconnectClientBySessionId("A123",
      inf.sessions[i].session_id);
   }
  }
 } catch (BrokerException ex) {
  System.out.println("Error while disconnecting BrokerClient by session\n"
    +ex);
  return;
 }
 .  .  .
```

Destroying Broker Clients

A specific Broker client may be destroyed by invoking the BrokerAdminClient.destroyClientById method.

The following example shows how to destroy a Broker client by ID:

```
.  .  .
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
   client_group, "My Broker Monitor",null);
 /* Destroy a particular Broker client */
 c.destroyClientById("A123");
 .  .  .
} catch (BrokerException ex) {
 System.out.println("Error while destroying BrokerClient\n"+ex);
 return;
}
.  .  .
```

Client State Sharing

The BrokerAdminClient.setClientStateShareLimitById method can be used to set the state sharing limit of a Broker client with a specific client identifier. webMethods Broker allows Broker clients in different applications to share the same client state. Sharing client state allows several Broker clients, possibly running on different hosts, to handle events in a parallel, first-come, first-served basis. This provides both parallel processing and load balancing for event handling.

Broker clients sharing the same client state are treated as one Broker client with regard to the state they are sharing. Any changes to the event subscriptions or event queue, such as clearing the queue, will affect all of the clients sharing the state.

See the *webMethods Broker Client Java API Programmer's Guide* for a thorough discussion of client state sharing.

Setting the Last-published Sequence Number

You can use the BrokerAdminClient.setClientLastPublishSequenceNumberById method to set the sequence number of the last event that the Broker client published. The Broker stores this sequence number and uses it to recognize and discard duplicate events which may be published by the Broker client.

Clearing a Broker Client's Queue

The BrokerAdminClient.clearClientQueueById method can be used to clear the event queue of a Broker client with a specific client identifier. See "Obtaining Broker Client Identifiers" on page 31 for information on discovering client identifiers at run time.

Important:
Use the BrokerAdminClient.clearClientQueueById method with care because it will delete events which have not yet been processed by the Broker client. If multiple clients are sharing the same client state, calling this method can have far-reaching effects.

You can use the BrokerAdminClient.getClientStatsById method to obtain information about a Broker client's queue, such as the current number of bytes and the number of events in the queue.

Tip:
To clear the event queues of all Broker clients on a Broker Server, use BrokerServerClient.pruneServerLog method.

Broker Client Subscriptions

Listing Subscriptions

You can use the BrokerAdminClient.getClientSubscriptionsById method to get a list of all subscriptions registered for a Broker client with a specific client identifier. An array of BrokerSubscription objects is returned that represents all the Broker client's subscriptions.

Testing the Existence of a Subscription

The BrokerAdminClient.doesClientSubscriptionExistById method can be used to determine if a subscription exists for a particular Broker client. To use this method you must specify the client identifier and the event type name. You must also specify a subscription filter if the subscription that you wish to match was created with a filter. The method will return `true` if the subscription exists.

Creating Subscriptions

The BrokerAdminClient.createClientSubscriptionById method can be used to register a subscription for the Broker client with a specified client identifier. You provide the client identifier and a BrokerSubscription object that describes the event subscription to be registered.

The BrokerAdminClient.createClientSubscriptionsById method can be used to create several event subscriptions for a Broker client in one method invocation. You provide the client identifier and an array of BrokerSubscription objects that describe the event subscriptions to be registered.

Note:
If any subscription specified by either of these methods has already been registered for the specified Broker client, the method will have no effect on that subscription.

Destroying Subscriptions

Use the BrokerAdminClient.destroyClientSubscriptionById method to destroy a subscription for the Broker client with a specified client identifier. You provide the client identifier and a BrokerSubscription object that describes the event subscription to be destroyed.

The BrokerAdminClient.destroyClientSubscriptionsById method can be used to destroy several event subscriptions for a Broker client in one method invocation. You provide the client identifier and an array of BrokerSubscription objects that describes the event subscriptions to be destroyed.

Getting Broker Client Statistics

The BrokerAdminClient.getClientStatsById method allows you to obtain statistics for a Broker client with a particular client identifier. The statistics are returned as fields within a BrokerEvent. You can retrieve each field using the BrokerEvent.get<type>Field method, described in the *webMethods Broker Client Java API Programmer's Guide*. The version of the get<type>Field method to be used will depend on the data type of field you are retrieving.

The following example illustrates how to obtain the system usage statistics and extract the value of two fields. In this example, the client identifier is hard-coded. See "Obtaining Broker Client Identifiers" on page 31 for information on discovering client identifiers at run time.

```
.  .  .
int events_published;
int events_retrieved;
BrokerEvent e;
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
   client_group, "My Broker Monitor",null);
 /* Get the BrokerEvent containing Broker client stats */
 e = c.getClientStatsById("A123");
 /* Get the number of events the Broker client has published */
 events_published = e.getIntegerField( "numEventsPublished");
 /* Get the number of events the Broker client has retrieved*/
 events_retrieved = e.getIntegerField( "numEventsRetrieved");
 .  .  .
} catch (BrokerException ex) {
 System.out.println("Error while getting Broker statistics\n"+ex);
 return;
}
.  .  .
```

Getting Broker Client Infosets

The BrokerAdminClient.getClientInfosetById method allows you to obtain the infoset for a particular Broker client. The infoset is returned as Broker event for convenience.

You can use the BrokerAdminClient.setClientInfosetById method to set the infoset for a Broker client, given its client ID.

4 Managing Client Groups

Overview

This chapter describes how you can use the BrokerAdminClient interface to obtain information about client groups defined for a Broker. Reading this chapter should help you understand:

- How to list client groups defined for a particular Broker.
- How to list all Broker clients belonging to a client group.
- How to obtain statistics for client groups defined for a Broker.
- How to create or destroy a client group.
- How to get or set a client group's properties, including event subscription and publication permissions.

Note:"Managing Broker Clients" on page 25 describes how to create, disconnect, reconnect and destroy a BrokerAdminClient object. For more information on client groups and their features, see *webMethods Broker Client Java API Programmer's Guide*.

Listing Client Groups

The BrokerAdminClient.getClientGroupNames method lets you obtain a list of all client group names known to the Broker to which your BrokerAdminClient is connected. This method is available for use by any BrokerAdminClient.

The following example illustrates how to list all client group names for a Broker:

```
.  .  .
int i;
String client_groups[];
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* List of all client groups for this Broker and print them out */
 client_groups = c.getClientGroupNames();
 for(i = 0;  i < client_groups.length;  i++) {
  System.out.println(client_groups[i]);
 }
} catch (BrokerException ex) {
 System.out.println("Error while getting the client group names\n"+ex);
 return;
}
.  .  .
```

Listing Client Identifiers by Group

The BrokerAdminClient.getClientIdsByClientGroup method lets you obtain a list of all client identifiers belonging to a particular client group.

The following example illustrates how to list client identifiers by client group:

```
.  .  .
int i;
String client_ids[];
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* Get a list of all client IDs for a client and print them out */
 client_ids = c.getClientIdsByClientGroup(“myGroup”);
 for(i = 0;  i < client_ids.length;  i++) {
  System.out.println(client_ids[i]+”\n”);
 }
} catch (BrokerException ex) {
 System.out.println("Error while getting the client IDs by group\n"+ex);
 return;
}
.  .  .
```

Getting Client Group Statistics

The BrokerAdminClient.getClientGroupStats method lets you obtain statistics for a client group. The statistics are returned as fields within a BrokerEvent.

After you have obtained the BrokerEvent containing the statistics, you can retrieve each field using the BrokerEvent.get<type>Field method, described in the *webMethods Broker Client Java API Programmer's Guide*. The version of the get<type>Field method to be used will depend on the data type of field you are retrieving. The following example illustrates how to obtain a BrokerEvent containing a client group's statistics and how to extract the value of two fields.

```
.  .  .
int events_published;
BrokerDate create_time;
BrokerEvent e;
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
   client_group, "My Broker Monitor",null);

 /* Get the BrokerEvent containing the client group statistics */
 e = c.getClientGroupStats(“myGroup”);
 /* Get the time and date when the client group was created */
 create_time = e.getDateField( “createTime”);
 /* Get the number of events published by all clients in this group */
 events_published = e.getIntegerField( “numEventsPublished”);
 .  .  .
} catch (BrokerException ex) {
 System.out.println("Error while getting the client group statistics\n"+ex);
 return;
}
.  .  .
```

Creating and Destroying Client Groups

You can use the BrokerAdminClient.createClientGroup method to create a client group for the Broker to which your BrokerAdminClient is connected. When creating a client group, you must specify the following:

- The Broker client group's name. For information on client group name restrictions, see the *webMethods Broker Client Java API Programmer's Guide*.
- The life cycle of the Broker client belonging to the group; see "Life Cycle" on page 42.
- The queue storage type for the Broker clients belonging to this group; see "Storage Type" on page 43.

The following example illustrates how you can create a client group using the BrokerAdminClient class.

```
.  .  .
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* Create a client group */
 c.createClientGroup("myGroup",
  BrokerAdminClient.LIFECYCLE_DESTROY_ON_DISCONNECT,
  BrokerTypeDef.STORAGE_GUARANTEED);
} catch (BrokerException ex) {
 System.out.println("Error while creating client group\n"+ex);
 return;
}
.  .  .
```

The BrokerAdminClient.setEventTypesAndClientGroups method can also be used to set event types as well as client groups for a Broker.

Life Cycle

The *life cycle* associated with your Broker client's client group will determine how long the Broker will maintain the *client state* for a Broker client belonging to that group. The life cycle is specified as one of the `BrokerAdminClient.LIFECYCLE_*` values shown in the following table.

Value	Description
`LIFECYCLE_DESTROY_ON_DISCONNECT`	The Broker will destroy the client state whenever the connection between a Broker client and the Broker is lost.
`LIFECYCLE_EXPLICIT_DESTROY`	The client state will be maintained if the Broker client is disconnected. Used for client applications that need to maintain state in the event of network or system failures.

Storage Type

The queue *storage type* associated with your Broker client's client group will determine the storage reliability of the events that are queued for a Broker client belonging to the group. The storage type is specified as one of the `BrokerTypeDef.STORAGE_*` values shown below.

Life Cycle	Description
`STORAGE_GUARANTEED`	The Broker will use a two-phase commit process to queue events for your Broker client. This offers the lowest performance, but very little risk of losing events if a hardware, software, or network failure occurs.
`STORAGE_VOLATILE`	The Broker will queue events for your Broker client using local memory. This offers higher performance along with a greater risk of losing events if a hardware, software, or network failure occurs.

Setting Client Group Descriptions

You can use the BrokerAdminClient.setClientGroupDescription method to set the description for a particular client group. You must supply the client group's name and the new description and your client must have permission to change the client group's information.

Note:
For information on client group name restrictions, see the *webMethods Broker Client Java API Programmer's Guide*.

Destroying Client Groups

The BrokerAdminClient.destroyClientGroup method allows your applications to destroy a client group defined for a particular Broker. If any existing Broker clients belong to the group, a BrokerDependencyException exception will be thrown and the client group will not be destroyed.

You can use the BrokerAdminClient.getClientIdsByClientGroup method to obtain a list of any Broker clients currently belonging to a group.

The following example illustrates how to destroy a client group for a Broker:

```
.  .  .
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* Destroy a client group and any current clients, too.  */
 c.destroyClientGroup(“myGroup”, true);
} catch (BrokerException ex) {
 System.out.println("Error while destroying client group\n"+ex);
 return;
}
.  .  .
```

BrokerClientGroupInfo

You can use the BrokerAdminClient.getClientGroupInfo method to obtain a BrokerClientGroupInfo object for a client group. The BrokerClientGroupInfo method contains the following client group information:

- Group name
- Group description
- Life cycle
- Storage type

The BrokerAdminClient.getClientGroupInfos method allows you to obtain BrokerClientGroupInfo objects for more than one client group.

Controlling Event Publications

The BrokerAdminClient.getClientGroupCanPublishList method lets you obtain all of the event types that may be published or delivered by Broker clients belonging to a particular client group.

The following example illustrates how to get a client group's can-publish list:

```
.  .  .
int i;
String publish_events[];
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* Get event types that may be published by clients in this group.*/
 publish_events = c.getClientGroupCanPublishList(“myGroup”);
 for(i = 0;  i < publish_events.length;  i++) {
  System.out.println(publish_events[i]);
 }
} catch (BrokerException ex) {
 System.out.println("Error while getting can-publish list\n"+ex);
 return;
}
.  .  .
```

Use the BrokerAdminClient.setClientGroupCanPublishList method to set the list of event types to which Broker clients belonging to a particular client group may subscribe.

The BrokerAdminClient.getClientGroupsWhichCanPublish method lets you obtain the names of all the client groups that can publish a particular event type.

The following example illustrates how to get client groups that can publish an event type:

```
int i;
String groups[];
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
```

```
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* Get a list of all client groups that may publish a
  * particular event type.
  */
 groups = c.getClientGroupsWhichCanPublish(“myEventType”);
 for(i = 0;  i < groups.length;  i++) {
  System.out.println(groups[i]);
 }
} catch (BrokerException ex) {
 System.out.println("Error while getting client groups\n"+ex);
 return;
}
.  .  .
```

Controlling Event Subscriptions

The BrokerAdminClient.getClientGroupCanSubscribeList method lets you obtain all of the event types to which Broker clients belonging to a particular client group my subscribe or receive as a delivered event.

The following example illustrates how to get a client group's can-subscribe list:

```
.  .  .
int i;
String subscribe_events[];
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* Get event types to which Broker clients in
  * this client group may subscribe.
  */
 subscribe_events = c.getClientGroupCanSubscribeList(“myGroup”);
 for(i = 0;  i < subscribe_events.length;  i++) {
  System.out.println(subscribe_events[i]);
 }
} catch (BrokerException ex) {
 System.out.println("Error while getting the can-subscribe list\n"+ex);
 return;
}
.  .  .
```

Use the BrokerAdminClient.setClientGroupCanSubscribeList method to set the list of event types to which Broker clients belonging to a particular client group may subscribe.

The BrokerAdminClient.getClientGroupsWhichCanSubscribe method lets you obtain the names of all the client groups that may subscribe to a particular event type or receive a delivered event of that type.

The following example illustrates how to get client groups that can subscribe to an event type:

```
.  .  .
int i;
String groups[];
BrokerAdminClient c;
```

```
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* Get a list of all client groups that may subscribe to
  * particular event type.
  */
 groups = c.getClientGroupsWhichCanPublish(“myEventType”);
 for(i = 0;  i < groups.length;  i++) {
  System.out.println(groups[i]);
 }
} catch (BrokerException ex) {
 System.out.println("Error while getting the client groups\n"+ex);
 return;
}
.  .  .
```

Controlling Group Access

For information on controlling security configurations, see “Managing Security Configurations” on page 55.

5 Managing Broker Event Types

Overview

This chapter describes how you can use the BrokerAdminClient interface, introduced in "Managing Broker Clients" on page 25, to obtain information about event types defined for a Broker. Reading this chapter should help you understand:

- Listing event types defined for a particular Broker.
- Listing all Broker clients which have subscriptions for an event type.
- Obtaining event type statistics.
- Creating or destroying an event type definition or event type infoset.
- Getting or setting a event type's properties, including event fields and their types.

Note:
For more information on event types and their features, see the *webMethods Broker Client Java API Programmer's Guide*.

Understanding the BrokerAdminTypeDef

Many of the BrokerAdminClient methods described in this chapter use BrokerAdminTypeDef objects as a parameter or a returned value. The BrokerTypeDef class provides your client application with a read-only description of an event type definition. In contrast, the BrokerAdminTypeDef class allows your client application to both query and modify the properties of an event type definition.

Note:
Unlike BrokerTypeDef objects, BrokerAdminTypeDef objects are not cached and are not affected if the client application's type definition cache is flushed.

BrokerAdminTypeDef objects are also used to represent the fields within an event definition. The following table shows the methods offered by the BrokerAdminTypeDef class.

Method	Description
clearField clearFields	Clears one or all fields in the type definition.
renameField	Renames a field within a BrokerAdminTypeDef object.
clearModificationFlag hasBeenModified setModificationFlag	Manages the modification flag, which is used to track changes made to the event type definition.
getBaseTypeName	Returns the event type name, without any event scope qualifier.
getScopeTypeName	Returns the event type's scope name.

Method	Description
getTypeName	Returns the event type's fully-qualified name.
setTypeName	Sets the fully-qualified name for the event type.
getDescription setDescription	Manages the description for the event type.
getFieldDef setFieldDef	Given a field name, obtains or sets the field's definition.
getFieldNames	Returns the names of the fields within an event or structure field.
getFieldType setFieldType	Manages the data type of an event field's value.
getStorageType setStorageType	Manages the event's storage type property.
getTimeToLive setTimeToLive	Manages the event's time-to-live property, which determines how long an event type will be available after being published before it expires and is destroyed.
insertFieldDef	Adds a field to an event type or a structure field.
isSystemDefined	Returns `true` if the event type is a system-defined type that may not be altered.
orderFields	Manages the order of the fields in an event type or structure field.
toString	Returns a string containing the type definition administration object in a human-readable format.

Listing Event Type Definitions

The BrokerClient.getEventTypeNames method can be used to return all the event type names defined for a Broker. Once you know the name of an event, you can use the BrokerAdminClient.getEventAdminTypeDef method to obtain an event type's definition. Your client must have permission to obtain this information.

The following example illustrates how to get an event type definition:

```
.  .  .
String event_names[];
BrokerAdminTypeDef def;
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
```

```
    client_group, "My Broker Monitor",null);
 /* Get a list of all event type */
 event_names = c.getEventTypeNames();
 /* Get the event definition for the first event type    */
 def = c.getEventAdminTypeDef(event_names[0]);
 System.out.println(“Event “+event_names[0]+” time-to-live=”
   def.getTimeToLive()+” seconds”);
} catch (BrokerException ex) {
 System.out.println("Error while getting event type definition\n"+ex);
 return;
}
.  .  .
```

Getting Multiple Event Type Definitions

The following example illustrates how you can use the BrokerAdminClient.getEventAdminTypeDefs method to obtain multiple event type definitions in one method invocation.

```
.  .  .
int i;
String event_names[];
BrokerAdminTypeDef defs[];
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);
 /* Get a list of all event types */
 event_names = c.getEventTypeNames();
 /* Get the event definition for each event type */
 defs = c.getEventAdminTypeDefs(event_names);
 for(i = 0;  i < defs.length;  i++) {
  /* Print the event def name and time-to-live value.  */
  System.out.println(“Event “+event_names[i]+” time-to-live=”
   defs[i].getTimeToLive()+” seconds”+”\n”);
 }
} catch (BrokerException ex) {
 System.out.println("Error while getting the event type definitions\n"+ex);
 return;
}
.  .  .
```

Getting Event Type Definitions by Scope

The BrokerAdminClient.getEventAdminTypeDefsByScope method allows your client application to obtain all the event type definitions belonging to a particular event *scope* in a single method invocation.

Getting Event Type Statistics

The BrokerAdminClient.getEventTypeStats method lets your client application obtain statistics for a particular event type definition. The statistics are returned as fields within a BrokerEvent.

After you have obtained the BrokerEvent containing the statistics, you can retrieve each field using the BrokerEvent.get<type>Field method, described in the *webMethods Broker Client Java API Programmer's*

Guide. The version of the get<type>Field method to be used will depend on the data type of field you are retrieving. The following example illustrates how to obtain a BrokerEvent containing an event type definition's statistics and extract the value of two fields.

```
.  .  .
BrokerDate create_time;
BrokerDate update_time;
BrokerEvent e;
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
   client_group, "My Broker Monitor",null);

 /* Get the BrokerEvent containing an event type's statistics */
 e = c.getEventTypeStats("myEventType");
 /* Get the time and date the event type definition was created */
 create_time = e.getDateField( "createTime");
 /* Get the time and date the event type definition was modified */
 update_time = e.getDateField( "updateTime");
} catch (BrokerException ex) {
 System.out.println("Error while getting the event type statistics\n"+ex);
 return;
}
.  .  .
```

Creating and Modifying Event Type Definitions

Your client application can use the BrokerAdminClient.setEventAdminTypeDef method to modify an event type's definition. Your application must first obtain and modify (or create and initialize) the desired BrokerAdminTypeDef object and provide it as a parameter to this method.

If the event type represented by the BrokerAdminTypeDef object does not exist on the Broker, it will be created. If the event type already exists on the Broker, it will be updated.

The BrokerAdminClient.setEventTypesAndClientGroups method can also be used to set event types as well as client groups for a Broker.

Modifying Event Fields

You can use the BrokerAdminTypeDef.insertFieldDef method to insert a new event field in an administrative type definition.

The BrokerAdminTypeDef.orderFields method allows you to change the order of event fields within an administrative type definition.

The BrokerAdminTypeDef.clearField method and the BrokerAdminTypeDef.clearFields method allow you to remove one or more event fields from an administrative type definition.

The BrokerAdminTypeDef.renameField method allows you to rename an event field within an administrative type definition.

You can use the BrokerAdminTypeDef.setFieldDef method to set the definition of an event field in an administrative type definition. If the specified field does not exist, it will be created.

The BrokerAdminTypeDef.setFieldType method allows you to set the data type of an event field in an administrative type definition. If the specified field does not exist, it will be created.

The following example illustrates how to create or modify an event type definition:

```
.  .  .
BrokerAdminTypeDef def;
BrokerAdminTypeDef field1, field2;
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
    client_group, "My Broker Monitor",null);

 /* Create the BrokerAdminTypeDef object for the event type */
 def = new BrokerAdminTypeDef(“myEventType”,
     BrokerTypeDef.FIELD_TYPE_EVENT);

 /* Set two fields in the event type definition */
 field1 = new BrokerAdminTypeDef(BrokerTypeDef.FIELD_TYPE_STRING);
 field2 = new BrokerAdminTypeDef(BrokerTypeDef.FIELD_TYPE_BOOLEAN);

 /* Add the fields to the event type definition */
 def.setFieldDef(“Name”, field1);
 def.setFieldDef(“Active”, field2);

  /* Set the event type definition.
   * Note that if it doesn’t exist, it will be created.
   * If the event type definition does exist, it will be
   * replaced with the new version.
   */
 c.setEventAdminTypeDef(def);
} catch (BrokerException ex) {
 System.out.println("Error while creating/modifying the event type def\n"+ex);
 return;
}
.  .  .
```

Creating or Modifying Multiple Event Type Definitions

The BrokerAdminClient.setEventAdminTypeDefs method allows your client application to modify or create several event type definitions in a single method invocation. Your application must first obtain and modify (or create and initialize) the desired BrokerAdminTypeDef objects and provide them as input to this method. If any of the BrokerAdminTypeDef objects you provide do not exist on the Broker, they will be created. If any of the event type definitions already exist on the Broker, they will be updated

Destroying Event Type Definitions

You can use the BrokerAdminClient.destroyEventType method to destroy an event type definition on the Broker to which your BrokerAdminClient is connected. If an existing Broker client subscription or client group uses the event type, a BrokerDependencyException exception will be thrown and the event type will not be destroyed.

You can use the following methods to determine which Broker clients or client groups are dependent on a particular event type:

- BrokerAdminClient.getClientIdsWhichAreSubscribed
- BrokerAdminClient.getClientGroupsWhichCanPublish
- BrokerAdminClient.getClientGroupsWhichCanSubscribe

The following example illustrates how to use the destroyEventType to destroy an event type definition:

```
.  .  .
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
   client_group, "My Broker Monitor",null);
 /* Destroy the event type definition using the force option */
 c.destroyEventType(“myEventType”, true);
} catch (BrokerException ex) {
 System.out.println("Error getting destroying event type\n"+ex);
 return;
}
.  .  .
```

Destroying Multiple Event Types

The BrokerAdminClient.destroyEventTypes method allows your client application to destroy several event type definitions in a single method invocation. You must supply the names of the event types you wish to destroy and an indication of whether to force the destroy operation if there are Broker clients currently using any of the event types.

Event Type Infosets

The following BrokerClient methods, which are described in the *webMethods Broker Client Java API Programmer's Guide*, allow you to retrieve event type infosets and infoset names:

- The BrokerClient.getEventTypeInfoset method can be used to retrieve a specific infoset for a particular event type.
- The BrokerClient.getEventTypeInfosets method can be used to retrieve all the infosets for a particular event type.
- The BrokerClient.getEventTypeInfosetNames method can be used to retrieve all the infoset names for a particular event type.

Setting Event Type Infosets

Your client application can use the BrokerAdminClient.setEventTypeInfoset method to modify an event type's infoset. Your application must first obtain and modify (or create and initialize) a BrokerEvent object that represents the infoset and provide it as a parameter to this method, along with the

event type name and infoset name. If your infoset name already exists for the event type name you specify, it will be updated. If it does not exist, it will be created.

Note:
For information on event type name and infoset name restrictions, see the *webMethods Broker Client Java API Programmer's Guide*.

Creating or Modifying Multiple Infosets

The BrokerAdminClient.setEventTypeInfosets method allows your client application to create or modify several infosets for a single event type definition in a one method invocation. You must supply an array of BrokerEvent objects that represent the infosets and the target event type name.

The name of each infoset will be taken from each BrokerEvent object's type name. If any of the infoset names already exist for the event type name you specify, they will be updated. If any of the infoset names do not exist, they will be created.

Destroying Infosets

You can use the BrokerAdminClient.destroyEventTypeInfoset method to destroy an infoset for an event type on the Broker to which your BrokerAdminClient is connected. You must supply the name of the event type and the name of the infoset you wish to destroy.

The following example illustrates how to use the destroyEventTypeInfoset method to destroy an infoset for an event type:

```
.  .  .
BrokerAdminClient c;
try {
 /* Create a Broker admin client */
 c = new BrokerAdminClient(broker_host, broker_name, null,
   client_group, "My ",null);
 /* Destroy an infoset for an event type definition */
 c.destroyEventTypeInfoset(“myEventType”, “myInfoset”);
} catch (BrokerException ex) {
 System.out.println("Error while destroying the infoset\n"+ex);
 return;
}
.  .  .
```

Destroying Multiple Infosets

The BrokerAdminClient.destroyEventTypeInfosets method allows your client application to destroy several infosets for a single event type in a single method invocation. You must supply the name of the event type and the names of the infosets you wish to destroy.

6 Managing Security Configurations

Overview

This chapter describes how your administrative client applications can use the Java interfaces to obtain information about and configure the security settings for a Broker Server, Broker, or client group.

Access Control Lists

The BrokerAccessControlList class is used to control the entities that may access a resource. A Broker Server can use an access control list (ACL) to determine which Broker clients are allowed administrative access. A client group may use an ACL to control who may create or reconnect a Broker client that is associated with the group.

A BrokerAccessControlList object actually contains two lists:

- **User list**
  - For basic authentication, a list of basic authentication user names.
  - For SSL authentication, a list of DNs of entities that are allowed access to the resource.
- **Authenticator list**
  - For basic authentication, a list of basic authentication aliases.
  - For SSL authentication, a list of DNs of certificate authorities that are trusted. These trusted authorities should be signers of the certificates associated with the user DNs.

An entity that wishes to access a resource associated with a BrokerAccessControlList must provide the credentials. Access is granted as follows:

- If both the user lists and authenticator lists are empty, access is granted.
- If the user list is not empty, the basic authentication user name or the entity's DN must be in the list.
- If the authenticator list is not empty, then the basic authentication alias or the entity's authenticator DN must be in the list.
- If both user lists and authenticator lists are not empty,
  - For basic authentication, the basic authentication user name must be on the user list and the basic authentication alias must be on the authenticator list.
  - For SSL authentication, the entity's DN must be on the user list and its authenticator DN must be on the authenticator list.

Getting and Setting Authenticator Names

The BrokerAccessControlList class offers methods for obtaining and setting the basic authentication aliases or the SSL DNs of trusted certificate authorities. An entity that wishes to access the resource associated with this BrokerAccessControlList object must provide the required credentials.

- You can use the BrokerAccessControlList.getAuthNames method to obtain the names of all the basic authentication aliases or SSL certificate authenticators.
- You can use the BrokerAccessControlList.getAuthNameState method to determine if a particular basic authentication alias or the SSL certificate authority's DN is defined in the list.
- The BrokerAccessControlList.setAuthNames method can be used to set all the basic authentication aliases or SSL DNs of trusted certificate authorities.
- You can use the BrokerAccessControlList.setAuthNameState method to add or remove a basic authenticator alias or SSL trusted certificate authority DN. You must provide basic authenticator alias or SSL DN and a boolean indicator signifying whether or not the name is allowed.
- The BrokerAccessControlList.setAuthNameStates method allows you to add or remove several basic authenticator aliases or SSL DNs with a single method invocation.

Getting and Setting Names

The BrokerAccessControlList class offers methods for obtaining and setting the user lists. This lists represent the entities that are allowed access to the resource associated with this BrokerAccessControlList object.

- You can use the BrokerAccessControlList.getUserNames method to obtain all the basic authentication user names or SSL user names.
- The BrokerAccessControlList.getUserNameState method can be used to determine if the specifies user is currently in the list.
- The BrokerAccessControlList.setUserNames method can be used to set all the basic authentication user names or SSL user names.
- You can use the BrokerAccessControlList.setUserNameState method to add or remove a basic authentication user or SSL user. You must provide a user name and a boolean indicator signifying whether or not the name is allowed.
- The BrokerAccessControlList.setUserNameStates method allows you to add or remove several basic authentication users or SSL users with a single method invocation.

Converting to a String

There are two BrokerAccessControlList.toString methods that allow you to convert the contents of an ACL to a string. One allows you to specify the number of 4-character spaces that are used for indentation.

Broker Server Security Configurations

A Broker Server's security configuration is used to control the Broker Server's identity, which is how the Broker Server is authenticated by the Broker clients that connect to them. The security configuration with which a Broker Server was started is called its active configuration. An administrative client application may change a Broker Server's saved security configuration, but

that configuration will not take effect until the next time the Broker Server is started. A Broker Server's active and saved configurations may be identical or they may be different, depending on whether or not the configuration has been changed since the Broker Server was started.

The BrokerConnectionDescriptor class contains the basic authentication user name and password information.

The BrokerSSLConfigV2 class contains the following SSL information:

- The names of the Broker Server's keystore and trust store files, and the password for accessing the keystore.
- The trusted root of the certificate authority that issued the Broker Server's certificate.
- The type of the SSL CRL file used by the SSL configuration and the path to the CRL file.

See *Administering webMethods Broker* for complete information on the use of keystore files, trust store files, and the information needed to configure SSL.

Important:
The BrokerSSLConfigV2 class only works with Broker Server version 7.1 and above; the related class, BrokerSSLConfig, only works with Broker Servers of version 6.5.2 and below.

Retrieving and Setting Broker Server Security Configurations

The following methods return the Broker Server's Identity:

- BrokerServerClient.getBasicAuthStatus returns `true` if basic authentication is configured on the Broker Server.
- BrokerServerClient.getSavedSSLidentity returns the Broker Server's basic authentication identity that is in use.
- BrokerServerClient.getActiveSSLIdentity and BrokerServerClient.getSavedSSLidentity return a BrokerSSLIdentity object
- BrokerSSLIdentity.getDN returns the Broker Server's SSL DN
- BrokerSSLIdentity.getIssuerDN returns the SSL authenticator DN for the Broker Server's SSL certificate.
- BrokerServerClient.getDNFromKeyStore and BrokerServerClient.getRootDNsFromTrustStore methods returns the SSL keystore and trust store information.
- BrokerServerClient.getSSLStatus returns the Broker Server's SSL status. The BrokerSSLStatus class describes whether SSL is enabled, disabled, not supported, or if there is an error in the configuration.

Retrieving and Setting Broker Server ACLs

The BrokerServerClient.getAdminACL method allows you to obtain a Broker Server's ACL. This list represents the user names or the authenticator names of all the entities that may issue administrative requests using a BrokerServerClient object.

You can use the BrokerServerClient.getDNsFromCertFile method to obtain all of the DNs from a Broker Server's certificate file.

You can use the BrokerServerClient.getRootDNsFromCertFile method to obtain all of the issuer DNs from a Broker Server's certificate file.

The BrokerServerClient.setAdminACL method allows you to set the ACL for a Broker Server.

Client Group Security Configurations

Client groups can use the BrokerAccessControlList class to determine which entities are allowed to create a BrokerClient or BrokerAdminClient object that belongs to the group.

You can use the BrokerAdminClient.getClientGroupACL method to obtain the ACL for a particular client group. You must supply the name of the client group whose ACL you wish to obtain.

The BrokerAdminClient.setClientGroupACL method allows you to set the ACL for a particular client group.

Obtaining Security Information for Client Sessions

You can determine which Broker client sessions are using basic authentication or SSL by following these steps:

1. Use the BrokerAdminClient.getClientInfoById or getClientInfosById method to obtain a BrokerClientInfo object for the particular Broker client in which you are interested.

2. Examine each BrokerClientSession object contained in the BrokerClientInfo.sessions data member.

3. Examine the BrokerClientSession.encrypt_level and encrypt_version data members to determine if SSL is being used. If SSL authentication is being used without encryption, then these data members will not be set. Note that BrokerClientSession.encrypt_level is deprecated.

4. Examine the BrokerClientSession.ssl_certificate data member to determine the Broker client's DN. While a Broker must always authenticate itself to a Broker client, the client is not required to authenticate itself to Broker. This data member will not be set under those circumstances.

Territory Security Configurations

Setting Territory Security

The BrokerAdminClient.setTerritorySecurity method allows you to set the level of security and the type of authentication that is to be used in the territory. All Brokers in the territory must be properly configured for the new settings and the changes may take some time to be propagated throughout the territory.

When using this method, you must provide the type of authentication to be used, specified as one of the `BrokerAdminClient.AUTH_TYPE_*` values.

Important:
Use this method with care because changing the security setting might cause the territory to become partially disconnected. For example, setting SSL authentication or encryption when one or more Brokers are not configured for SSL will cause the territory to become partially disconnected.

Territory ACLs

Brokers in a territory can use the BrokerAccessControlList class to authenticate Brokers that want to join the territory. ACLs are only checked when a Broker joins the territory.

You can use the BrokerAdminClient.getTerritoryACL method to obtain the ACL for the Broker's territory.

You can use the BrokerAdminClient.setTerritoryACL method to configure the ACL for the Broker's territory.

Territory Gateway Security Configurations

Setting Territory Security

The BrokerAdminClient.setTerritoryGatewaySecurity method allows you to set the level of security and the type of authentication that is to be used in a territory gateway. Each of the two Brokers that make up the gateway must be properly configured for the new settings.

When using this method, you must provide the following parameters:

- The territory name.
- The type of authentication to be used, specified as one of the `BrokerAdminClient.AUTH_TYPE_*` values.
- For information on territory gateway security, see *Administering webMethods Broker*.

Important:
Use this method with care because changing the security setting might cause the gateway to become disconnected. For example, setting SSL authentication or encryption when the remote Broker is not configured for SSL will cause the gateway to become disconnected.

Territory Gateway ACLs

Brokers that make up a territory gateway can use the BrokerAccessControlList class to authenticate the remote Broker that makes up the other half of the gateway. ACLs are only checked when a Broker opens a connection to the other Broker in the gateway.

You can use the BrokerAdminClient.setTerritoryGatewayACL method to configure the ACL for the territory gateway, and the BrokerAdminClient.getTerritoryGatewayACL method to obtain the ACL information for the gateway. In both cases, a BrokerException will be thrown if basic authentication or SSL is not enabled.

Cluster Security Configurations

Setting Cluster Security

The BrokerAdminClient.setClusterSecurity method allows you to set the level of security and the type of authentication that is to be used in the cluster. All Brokers in the cluster must be properly configured for the new settings and the changes may take some time to be propagated throughout the cluster.

When using this method, you must provide the following parameters:

- The type of authentication to be used, specified as one of the `BrokerAdminClient.AUTH_TYPE_*` values.

Important:
Use this method with care because changing the security setting might cause the cluster to become partially disconnected. For example, setting SSL authentication or encryption when one or more Brokers are not configured for SSL will cause the cluster to become partially disconnected.

Cluster ACLs

Brokers in a cluster can use the BrokerAccessControlList class to authenticate Brokers that want to join the cluster. ACLs are only checked when a Broker joins the cluster.

You can use the BrokerAdminClient.getClusterACL method to obtain the ACL for the Broker's cluster.

You can use the BrokerAdminClient.setClusterACL method to configure the ACL for the Broker's cluster.

Cluster Gateway Security Configurations

Setting Cluster Security

The BrokerAdminClient.setClusterGatewaySecurity method allows you to set the level of security and the type of authentication that is to be used in a cluster gateway. Each of the two Brokers that make up the gateway must be properly configured for the new settings.

When using this method, you must provide the following parameters:

- The cluster name
- The Broker name
- The type of authentication to be used, specified as one of the `BrokerAdminClient.AUTH_TYPE_*` values

For information on cluster gateway security, see *Administering webMethods Broker*.

Important:
Use this method with care because changing the security setting might cause the gateway to become disconnected. For example, setting SSL authentication or encryption when the remote Broker is not configured for SSL will cause the gateway to become disconnected.

Cluster Gateway ACLs

Brokers that make up a cluster gateway can use the BrokerAccessControlList class to authenticate the remote Broker that makes up the other half of the gateway. ACLs are only checked when a Broker opens a connection to the other Broker in the gateway.

You can use the BrokerAdminClient.setClusterGatewayACL method to configure the ACL for the cluster gateway, and the BrokerAdminClient.getClusterGatewayACL method to obtain the ACL information for the gateway. In both cases, a BrokerException will be thrown if basic authentication or SSL is not enabled.

7 Managing Broker Territories

Overview

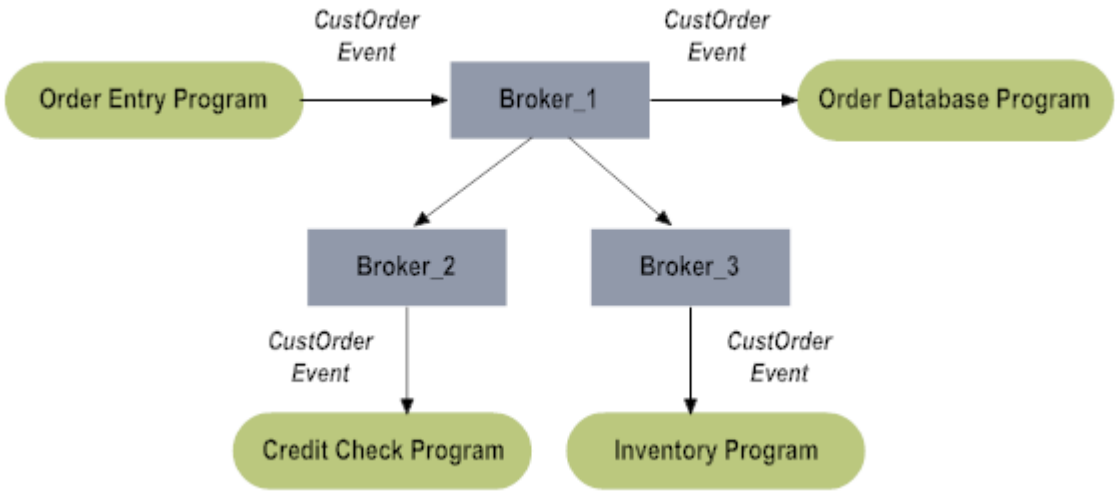
This chapter describes how your administrative client applications can create and manage Broker-to-Broker communication through territories and territory gateways.

Using Territories

The webMethods Broker systems allows two or more Brokers to share information about their event type definitions and client groups. This sharing of information enables communication between Broker clients connected to different Brokers. The figure below shows how an event published by a client program connected to `Broker_1` can be received by a client program connected to either `Broker_1`, `Broker_2`, or `Broker_3`.

Note:
A Broker can be a member of no more than one territory. However, territories can be linked together using territory gateways. For more information, see "Using Territory Gateways" on page 67.

Broker-to-Broker communication



To share information and forward events, Brokers must join a territory. A Broker can only belong to one territory and all Brokers within the same territory have knowledge of one another's event type definitions and client groups. For more information on this feature, see *Administering webMethods Broker*.

Creating a New Territory

The BrokerAdminClient.createTerritory method allows you to create a new territory. You must provide the name of the territory to be created.

Note:
For information on territory name restrictions, see the *webMethods Broker Client Java API Programmer's Guide*.

When the new territory is created, the Broker to which the administrative client is connected will become the first member of that territory.

Adding a Broker to an Existing Territory

You can use the BrokerAdminClient.joinTerritory method to add a Broker to an existing territory. You must provide the following arguments to this method:

- The name of the remote Broker whose territory this Broker is to join.
- The name of the Broker host where the remote Broker is running.

The Broker to which the administrative client is connected will be added to the remote Broker's territory.

Removing a Broker from a Territory

The BrokerAdminClient.leaveTerritory method lets you remove the Broker to which your BrokerAdminClient is connected from its territory. When a Broker leaves a territory, communication must take place between the Broker being removed and the other Brokers in the territory.

You must provide the following arguments to this method:

- The number of milliseconds to wait for the operation to complete. You can specify -1 to indicate you do not wish the operation to be subject to a time-out.
- An indication of whether or not the operation should be forced to complete, even if communication with the other Brokers in the territory exceeds the time limit.

You can use the BrokerAdminClient.removeBrokerFromTerritory method to remove a Broker from a territory. This function will not wait for the communication between the other Brokers in the territory to complete, but returns as soon as the removal process is initiated.

You must provide the following arguments to this method:

- The name of the Broker to be removed from the territory.
- The name of the Broker host where the Broker is executing.

Getting Territory Information

You can use the BrokerAdminClient.getTerritoryInfo method to obtain territory information for the Broker to which your Broker client is connected. A BrokerTerritoryInfo object is returned that contains the territory name, Broker host, Broker name, and Broker description for your Broker. An exception will be thrown if the Broker is not a member of a territory.

Listing Brokers in a Territory

The BrokerAdminClient.getBrokersInTerritory method allows you to obtain information about all the other Brokers in the territory to which the Broker client's Broker belongs.

The array of BrokerInfo objects returned are used to contain the territory name, Broker host, Broker name, and Broker description for a particular Broker.

Obtaining Territory Statistics

You can use the BrokerAdminClient.getTerritoryStats method to obtain statistics about the territory to which the Broker client's Broker belongs. These statistics include information about the other Brokers in the territory as well as information about the number of events that have been enqueued, forwarded, and received.

The statistics are returned as a BrokerEvent, for convenience.

Maintaining a Connection Between Brokers in a Territory

In network configurations where territory Brokers are separated by a firewall, the firewall may terminate a connection if the connection remains idle for a long period of time. To prevent this from occurring, you can use the BrokerAdminClient.setRemoteBrokerKeepAlive method. This method causes a remote Broker to periodically issue "keep-alive" messages to another Broker in its territory, preventing the connection between them from remaining idle too long.

Activating the Keep-Alive Feature between Territory Brokers

To activate the keep-alive feature between two Brokers in a territory, you use the BrokerAdminClient.setRemoteBrokerKeepAlive method and specify the following two parameters:

- *broker_name*. The name of the Broker to which you want the remote Broker to issue keep-alive messages.
- *KeepAlive*. The number of seconds between the keep-alive messages. To disable gateway the keep-alive feature, specify 0 seconds.

Be aware that BrokerAdminClient.setTerritoryGatewayKeepAlive method activates the keep-alive feature in one direction. That is, it instructs the Broker to which the BrokerAdminClient client is connected, to issue keep-alive messages to a second Broker in the territory. It does not instruct the second Broker to issue keep-alive messages. To activate the keep-alive feature in both directions, you must execute BrokerAdminClient.setRemoteBrokerKeepAlive on both Brokers.

To implement the keep-alive feature correctly, you must understand the conditions that cause the firewalls to drop a connection. Given this information, you can determine which Broker should issue keep-alive messages (it may be both) and the frequency at which the messages need to be issued to prevent the connection from dropping.

Checking the Keep-Alive Settings for a Broker

To get the Broker's current keep-alive setting, you use the BrokerAdminClient.getRemoteBrokerKeepAlive method. This method returns the remote Broker's current keep-alive interval (in seconds) for a connection to another Broker in its territory.

Setting Territory Security

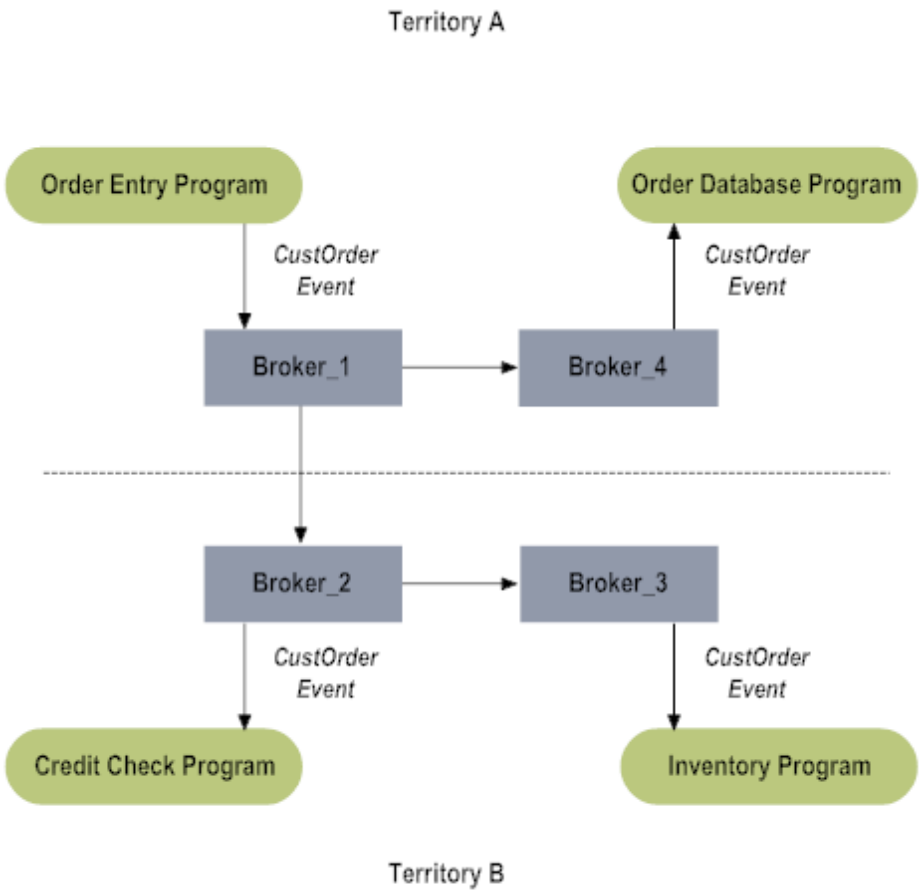
For information on territory security, see "Territory Security Configurations" on page 60.

Using Territory Gateways

Though the webMethods Broker systems restricts a Broker to membership in more than one territory at a time, you can configure a *territory gateway* to enable event forwarding between different territories. A Broker may act as a gateway to a single territory or to multiple territories.

The next figure shows a territory gateway between `Broker_1` and `Broker_2` that effectively joins the two territories. The territory gateway allows an event published by a client program connected to `Broker_1`, a member of territory A, to be forwarded through the territory gateway and be received by a client program connected to `Broker_3`, a member of territory B.

Territory gateway communication



Creating and Destroying Gateways

The BrokerAdminClient.createTerritoryGateway method allows you to obtain information about all the other Brokers in the territory to which the Broker client's Broker belongs.

You must provide the following arguments to this method:

- The name of the remote Broker that will act as the other endpoint of the gateway.
- The name of the host where the remote Broker is running.
- The name of the remote Broker's territory.

A territory gateway consists of two endpoints, so this function must be invoked for both endpoints before the gateway activation complete. Failure to activate both ends of the gateway will result in the gateway being in an incomplete state.

You can use the BrokerAdminClient.destroyTerritoryGateway method to destroy one endpoint of a territory gateway. You must provide the following arguments to this method:

- Name of the Broker to which this gateway is connected in the specified territory.
- The name of the territory whose gateway is to be destroyed.

A territory gateway consists of two endpoints, so this function must be invoked for both endpoints before the gateway destruction is complete.

Setting Shared Event Types

The BrokerAdminClient.setTerritoryGatewaySharedEventTypes method can be used to set the event types that will be forwarded or received by a gateway Broker. You specify the following parameters:

- The territory name.
- An array of BrokerSharedEventTypeInfo objects.

Each BrokerSharedEventTypeInfo object contains:

- The event type name.
- An indication of whether or not this Broker will accept events of this type that are being published or delivered from the remote territory
- An indication of whether or not this Broker will accept forwarded subscriptions or deliver events of this type to the remote territory.
- An indication of whether or not this event type is synchronized with the remote territory.

Listing Local Gateways

The BrokerAdminClient.getLocalClusterGateways method lets you obtain information about all the gateways available from the current Broker. An array of BrokerTerritoryGatewayInfo objects are returned which describe the gateways.

Listing All Gateways

You can use the BrokerAdminClient.getAllClusterGateways method to obtain information about all the gateways available within your Broker's territory or through gateways accessible to your Broker. An array of BrokerTerritoryGatewayInfo objects are returned which describe the gateways.

Managing Event Types

The BrokerAdminClient.getTerritoryGatewaySharedEventTypes method lets you obtain information about all the event types that are shared between two territory gateway Brokers. You must provide the following arguments to the territory name whose shared event type information is to be returned. This method returns an array of BrokerSharedEventTypeInfo objects that describes all of the shared event types.

The BrokerAdminClient.setTerritoryGatewaySharedEventTypes method allows you to set the complete list of shared event types for a territory gateway. The `is_synchronized` variable in each element of infos will be ignored.

Note:
To configure gateway event type sharing, first save the can-subscribe settings on both sides of the gateway and then save the can-publish settings. It is important to set the can-subscribe settings before the can-publish settings and to set them in separate invocations of this function.

You can control the content of the events that are forwarded through the gateway by specifying a `subscribe_filter`. If event filtering by the gateway is not desired, the `subscribe_filter` member should be set to `null`. For more information of event filtering, see the *webMethods Broker Client Java API Programmer's Guide*.

Note:
When porting applications from an earlier release, be sure to set the subscribe filter to either a valid value or to `NULL`.

Obtaining Gateway Statistics

You can use the BrokerAdminClient.getTerritoryGatewayStats method to obtain statistics about a territory gateway. These statistics include information about the gateway Broker as well as information about the number of events that have been enqueued, forwarded, and received.

You must provide the name of the territory whose statistics are to be returned.

The statistics are returned as an event, for convenience.

Maintaining a Connection Between Gateway Brokers

In network configurations where gateway Brokers are separated by a firewall, the firewall may terminate a connection if the connection remains idle for a long period of time. To prevent this from occurring between gateway Brokers, you can use the BrokerAdminClient.setTerritoryGatewayKeepAlive method. This method causes a gateway Broker to periodically issue "keep-alive" messages to the other Broker, preventing the connection between them from remaining idle too long.

Activating the Keep-Alive Feature between Gateway Brokers

To activate the keep-alive feature on a gateway Broker, you use the BrokerAdminClient.setTerritoryGatewayKeepAlive method and specify the following two parameters:

- *territory_name*. The name of the territory where the remote gateway Broker resides.
- *KeepAlive*. The number of seconds between the keep-alive messages. To disable gateway keep-alive feature, specify 0 seconds.

Be aware that BrokerAdminClient.setTerritoryGatewayKeepAlive method activates the keep-alive feature in one direction. That is, it instructs the Broker to which the BrokerAdminClient client is connected, to issue keep-alive messages to the gateway Broker in the other territory. It does not instruct the remote gateway Broker to issue keep-alive messages. To activate keep-alive messages in both directions, you must execute BrokerAdminClient.setTerritoryGatewayKeepAlive on both gateway Brokers.

To implement the keep-alive feature correctly, you must understand the conditions that cause the firewalls to drop a connection. (In most cases, you will need to coordinate with the network administrators at each end of the gateway to obtain information about the behavior of the firewalls.) Given this information, you can determine which gateway Broker must issue keep-alive messages (it may be both) and the frequency at which the messages need to be issued to prevent the connection from dropping.

Checking the Keep-Alive Settings for a Gateway Broker

To get the gateway Broker's current keep-alive setting for a particular territory, you use the BrokerAdminClient.getTerritoryGatewayKeepAlive method. This method returns the gateway Broker's current keep-alive interval (in seconds) for a given territory.

Pausing a Gateway Broker

To temporarily halt forwarding of events from one gateway Broker to another, you use the BrokerAdminClient.setTerritoryGatewayPause method. When you use this method, you specify the territory whose events you want to hold and set the *pause* parameter to `true`.

Once you pause the forwarding of events to a particular territory, the gateway Broker queues documents for that territory in an event-forwarding queue. The Broker does not resume forwarding events to the remote territory until event-forwarding is explicitly re-enabled.

Pausing Traffic in Both Directions

Be aware that the BrokerAdminClient.setTerritoryGatewayPause method halts the flow of traffic in one direction. That is, it stops the Broker to which the BrokerAdminClient client is connected from sending outbound messages to a specified territory. It does not halt the flow of inbound traffic to the gateway Broker from the remote territory. To halt both inbound and outbound between two gateway Brokers, you must execute BrokerAdminClient.setTerritoryGatewayPause on both Brokers.

Note that once you pause the forwarding queue for a territory, it remains paused even if the gateway Broker is restarted or the Broker Admin client disconnects.

Pausing Traffic to and from Multiple Territories

If a Broker serves as a gateway to multiple territories, and you want to pause outbound traffic to all of the territories, execute BrokerAdminClient.setTerritoryGatewayPause on the gateway Broker once for each remote territory.

Similarly, if you want to halt inbound traffic to a gateway Broker from multiple remote territories, execute BrokerAdminClient.setTerritoryGatewayPause on each remote gateway Broker.

Unpausing a Gateway Broker

To re-enable event forwarding, you re-execute BrokerAdminClient.setTerritoryGatewayPause and set the *pause* parameter to `false`.

Checking the Current Pause Setting

To check the state of event-forwarding queues on the gateway Broker, execute the BrokerAdminClient.getTerritoryStats method and examine the *gateway_pause* field in the Broker event that BrokerAdmin.getTerritoryStats returns.

Static Gateway Forwarding

The static gateway forwarding feature ensures consistent document flow across a gateway between two territories at all times. Documents defined by the gateway permissions list are allowed to flow over the gateway. Once static gateway forwarding is enabled, the documents are forwarded to the remote territory at all times, regardless of any client subscriptions on the remote territory.

Essentially, the static gateway forwarding feature force feeds documents to the remote territory. When there are no subscribers on the remote territory, forwarded documents will be dropped at the gateway peer of the remote territory. Disabling static gateway forwarding resumes the normal behavior of subscription based gateway forwarding.

Note:
The static gateway forwarding feature is only supported on a webMethods Broker Version 6.1 Service Pack 4 and later. The Brokers on the other end of the gateway can be of any version. In addition, you cannot enable static gateway forwarding on a Broker that is either not in a territory or does not have a gateway to the specified remote territory.

To enable static gateway forwarding, use the BrokerAdminClient.setStaticGatewayForwarding method and set *enable* to true.

To check the status of the static gateway subscription set on a remote territory from the current Broker, execute the BrokerAdminClient.getStaticGatewayForwarding method.

For more information about static gateway forwarding, see *Administering webMethods Broker*.

Refusing Document Type Updates Across a Gateway

By default, changes made to shared document types on one side of a gateway are propagated to a Broker territory on the other side of the gateway. However, you can configure a territory to refuse updates made on the other side of a gateway. This setting provides the following security measures:

- Lock down of a local territory. Unauthorized changes made in a neighboring territory are not carried over to the local territory.
- Protection against accidental changes made to shared document types in the neighboring territory.

Configuring a territory to refuse updates is done at the local end of a gateway. If you want the Broker territories on both sides of a gateway to refuse updates, you must configure the setting from both sides.

Update Refusal Classes and Methods

You have the option of refusing all changes to document types made in the neighboring territory, or specifying which document types cannot be changed.

- The following methods set and return whether a gateway refuses all updates made in a neighboring territory: BrokerAdminClient.setTerritoryGatewayRefuseAllUpdates and BrokerAdminClient.getTerritoryGatewayRefuseAllUpdates.
- The following methods set and return the document types for which a gateway refuses updates: BrokerAdminClient.setTerritoryGatewayRefuseEventTypeUpdates and BrokerAdminClient.getTerritoryGatewayRefuseEventTypeUpdates.
- The BrokerSharedEventTypeRefuseUpdateInfo class stores information about how a particular event type is shared across a territory gateway, and whether the event type itself or updates to the event type are refused.
- If you configure both the BrokerAdminClient.setTerritoryGatewayRefuseAllUpdates and BrokerAdminClient.setTerritoryGatewayRefuseEventTypeUpdates methods, BrokerAdminClient.setTerritoryGatewayRefuseAllUpdates takes precedence, and updates to all shared document types are refused.

Synchronizing Updates

If you turned on an update refusal setting for the local territory and an update is made to a shared document type on the other side of the territory gateway, all document flow to the local territory stops. At this point, you need to synchronize the document types by using one of the methods described below:

- For update refusals set on the local side of a gateway, deliver a document of the type that was refused to the remote side of the gateway.
- Change the event type definition;
  - On the remote side of the gateway, revert the update back to the previous document type definition.
  - On the local side of the gateway, update the document type definition.
- Turn off the update refusal flag.

After you synchronize the updates, document flow to the local territory will restart.

Note: Broker Server issues alerts for event flow suspension and resumption.

Setting Territory Gateway Security

For information on territory gateway security, see "Territory Gateway Security Configurations" on page 60.

8 Managing Broker Clusters

Overview

This chapter describes how your administrative client applications can create and manage Broker-to-Broker communication through clusters and cluster gateways.

Using Clusters

The webMethods Broker systems allows two or more Brokers to share information about their event type definitions and client groups. This sharing of information enables communication between Broker clients connected to different Brokers.

Note:
A Broker may be a member of no more than one cluster. However, clusters can be linked together using cluster gateways. For more information, see "Using Cluster Gateways" on page 78.

To share information and forward events, Brokers must join a cluster. A Broker can only belong to one cluster and all Brokers within the same cluster have knowledge of one another's event type definitions and client groups. The Brokers in a cluster do not forward documents to other Brokers in the cluster. For more information on this feature, see *Administering webMethods Broker*.

Creating a New Cluster

The BrokerAdminClient.createCluster method allows you to create a new cluster. You must provide the name of the cluster to be created.

Note:
For information on cluster name restrictions, see the *webMethods Broker Client Java API Programmer's Guide*.

When the new cluster is created, the Broker to which the administrative client is connected will become the first member of that cluster.

Adding a Broker to an Existing Cluster

You can use the BrokerAdminClient.addBrokerToCluster method to add a Broker to an existing cluster. You must provide the following arguments to this method:

- The name of the remote Broker whose cluster this Broker is to join
- The name of the Broker host where the remote Broker is running

The Broker to which the administrative client is connected will be added to the remote Broker's cluster.

Removing a Broker from a Cluster

The BrokerAdminClient.leaveCluster method lets you remove the Broker to which your BrokerAdminClient is connected from its cluster. When a Broker leaves a cluster, communication must take place between the Broker being removed and the other Brokers in the cluster.

You must provide the following arguments to this method:

- The number of milliseconds to wait for the operation to complete. You can specify -1 to indicate you do not wish the operation to be subject to a time-out.
- An indication of whether or not the operation should be forced to complete, even if communication with the other Brokers in the cluster exceeds the time limit.

You can use the BrokerAdminClient.removeBrokerFromCluster method to remove a Broker from a cluster. This function will not wait for the communication between the other Brokers in the cluster to complete, but returns as soon as the removal process is initiated.

You must provide the following arguments to this method:

- The name of the Broker to be removed from the cluster.
- The name of the Broker host where the Broker is executing.

Getting Cluster Information

You can use the BrokerAdminClient.getClusterInfo method to obtain cluster information for the Broker to which your Broker client is connected. A BrokerClusterInfo object is returned that contains the cluster name, Broker host, Broker name, and Broker description for your Broker. An exception will be thrown if the Broker is not a member of a cluster.

Listing Brokers in a Cluster

The BrokerAdminClient.getBrokersInCluster method allows you to obtain information about all the other Brokers in the cluster to which the Broker client's Broker belongs.

The array of BrokerInfo objects returned are used to contain the cluster name, Broker host, Broker name, and Broker description for a particular Broker.

Obtaining Cluster Statistics

You can use the BrokerAdminClient.getClusterStats method to obtain statistics about the cluster to which the Broker client's Broker belongs. These statistics include information about the other Brokers in the cluster as well as information about the number of events that have been enqueued, forwarded, and received.

The statistics are returned as a BrokerEvent, for convenience.

Maintaining a Connection Between Brokers in a Cluster

In network configurations where cluster Brokers are separated by a firewall, the firewall may terminate a connection if the connection remains idle for a long period of time. To prevent this from occurring, you can use the BrokerAdminClient.setRemoteBrokerKeepAlive method. This method causes a remote Broker to periodically issue "keep-alive" messages to another Broker in its cluster, preventing the connection between them from remaining idle for too long.

Activating the Keep-Alive Feature between Cluster Brokers

To activate the keep-alive feature between two Brokers in a cluster, you use the BrokerAdminClient.setRemoteBrokerKeepAlive method and specify the following two parameters:

- *broker_name*. The name of the Broker to which you want the remote Broker to issue keep-alive messages.
- *KeepAlive*. The number of seconds between the keep-alive messages. To disable gateway the keep-alive feature, specify 0 seconds.

Be aware that BrokerAdminClient.setClusterGatewayKeepAlive method activates the keep-alive feature in one direction. That is, it instructs the Broker to which the BrokerAdminClient client is connected, to issue keep-alive messages to a second Broker in the cluster. It does not instruct the second Broker to issue keep-alive messages. To activate the keep-alive feature in both directions, you must execute BrokerAdminClient.setRemoteBrokerKeepAlive on both Brokers.

To implement the keep-alive feature correctly, you must understand the conditions that cause the firewalls to drop a connection. Given this information, you can determine which Broker should issue keep-alive messages (it may be both) and the frequency at which the messages need to be issued to prevent the connection from dropping.

Checking the Keep-Alive Settings for a Broker

To get the Broker's current keep-alive setting, you use the BrokerAdminClient.getRemoteBrokerKeepAlive method. This method returns the remote Broker's current keep-alive interval (in seconds) for a connection to another Broker in its cluster.

Setting Cluster Security

For information on cluster security, see “Cluster Security Configurations” on page 61.

Using Cluster Gateways

Though the webMethods Broker systems restricts a Broker to membership in more than one cluster at a time, you can configure a *cluster gateway* to enable event forwarding between different clusters. A Broker may act as a gateway to a single cluster or to multiple clusters. One Broker can have multiple gateway connections to the Brokers in the other cluster. Multiple gateway connections is not possible in territories.

Creating and Destroying Cluster Gateways

The BrokerAdminClient.createClusterGateway method creates a cluster gateway to the specified cluster by having this Broker communicate with the remote Broker specified by *broker_host* and *broker_name*.

You must provide the following arguments to this method:

- The name of the remote Broker's cluster
- The name of the host where the remote Broker is running
- The name of the remote Broker that will act as the other endpoint of the gateway

A cluster gateway consists of two endpoints, so this function must be invoked for both endpoints before the gateway activation is complete. Failure to activate both ends of the gateway will result in the gateway being in an incomplete state.

You can use the BrokerAdminClient.destroyClusterGateway method to destroy one endpoint of a cluster gateway. You must provide the following arguments to this method:

- The name of the cluster whose gateway is to be destroyed
- Name of the Broker to which this gateway is connected in the specified cluster

A cluster gateway consists of two endpoints, so this function must be invoked for both endpoints before the gateway destruction is complete.

Setting Shared Event Types

The BrokerAdminClient.setClusterGatewaySharedEventTypes method can be used to set the event types that will be forwarded or received by a gateway Broker. You specify the following parameters:

- The cluster name
- Broker name in the specified cluster
- An array of BrokerSharedEventTypeInfo objects

Each BrokerSharedEventTypeInfo object contains:

- The event type name
- An indication of whether or not this Broker will accept events of this type that are being published or delivered from the remote cluster
- An indication of whether or not this Broker will accept forwarded subscriptions or deliver events of this type to the remote cluster
- An indication of whether or not this event type is synchronized with the remote cluster

Setting Primary Gateway

The BrokerAdminClient.setClusterPrimaryGateway method can be used to set the primary gateway to the cluster. You specify the following parameters:

- The cluster name.
- Broker name belonging to the specified cluster.

> **Note:**
> Ensure that all the cluster gateway links between a pair of clusters are identical. The permissions such as the sharedEventTypeInfo and KeepAlive must be the same on all the gateway links. During gateway link failover, when the connection switches from the primary gateway connection to the next available gateway connection, the gateway connection setting must be same as the failed gateway link so that the switch over does not impact the runtime operations.

Listing Local Gateways

The BrokerAdminClient.getLocalClusterGateways method lets you obtain information about all the gateways available from the current Broker. An array of BrokerClusterGatewayInfo objects are returned which describe the gateways.

Listing All Gateways

You can use the BrokerAdminClient.BrokerAdminClient method to obtain information about all the gateways available within your Broker's cluster or through gateways accessible to your Broker. An array of BrokerClusterGatewayInfo objects are returned which describe the gateways.

Managing Event Types

The BrokerAdminClient.getClusterGatewaySharedEventTypes method lets you obtain information about all the event types that are shared between two cluster gateway Brokers. This method returns an array of BrokerSharedEventTypeInfo objects which describe all of the shared event types.

The BrokerAdminClient.setClusterGatewaySharedEventTypes method allows you to set the complete list of shared event types for a cluster gateway. The `is_synchronized` variable in each element of infos will be ignored.

> **Note:**
> To configure gateway event type sharing, first save the can-subscribe settings on both sides of the gateway and then save the can-publish settings. It is important to set the can-subscribe settings before the can-publish settings and to set them in separate invocations of this function.

You can control the content of the events that are forwarded through the gateway by specifying a `subscribe_filter`. If event filtering by the gateway is not desired, the `subscribe_filter` member should be set to `null`. For more information of event filtering, see the *webMethods Broker Client Java API Programmer's Guide*.

Note:
When porting applications from an earlier release, be sure to set the subscribe filter to either a valid value or to `NULL`.

Obtaining Gateway Statistics

You can use the BrokerAdminClient.getClusterGatewayStats method to obtain statistics about a cluster gateway. These statistics include information about the gateway Broker as well as information about the number of events that have been enqueued, forwarded, and received. You must provide the name of the cluster and the name of the cluster gateway Broker.

Maintaining a Connection Between Gateway Brokers

In network configurations where gateway Brokers are separated by a firewall, the firewall may terminate a connection if the connection remains idle for a long period of time. To prevent this from occurring between gateway Brokers, you can use the BrokerAdminClient.setClusterGatewayKeepAlive method. This method causes a gateway Broker to periodically issue "keep-alive" messages to the other Broker, preventing the connection between them from remaining idle too long.

Activating the Keep-Alive Feature between Gateway Brokers

To activate the keep-alive feature on a gateway Broker, you use the BrokerAdminClient.setClusterGatewayKeepAlive method and specify the following three parameters:

- *cluster_name*. The name of the cluster where the remote gateway Broker resides
- *Broker name*.The name of the Broker belonging to the cluster specified by the *cluster_name*.
- *KeepAlive*. The number of seconds between the keep-alive messages. To disable gateway keep-alive feature, specify 0 seconds.

Be aware that BrokerAdminClient.setClusterGatewayKeepAlive method activates the keep-alive feature in one direction. That is, it instructs the Broker to which the BrokerAdminClient client is connected, to issue keep-alive messages to the gateway Broker in the other cluster. It does not instruct the remote gateway Broker to issue keep-alive messages. To activate keep-alive messages in both directions, you must execute BrokerAdminClient.setClusterGatewayKeepAlive on both gateway Brokers.

To implement the keep-alive feature correctly, you must understand the conditions that cause the firewalls to drop a connection. (In most cases, you will need to coordinate with the network administrators at each end of the gateway to obtain information about the behavior of the firewalls.) Given this information, you can determine which gateway Broker must issue keep-alive messages (it may be both) and the frequency at which the messages need to be issued to prevent the connection from dropping.

Checking the Keep-Alive Settings for a Cluster Gateway Broker

To get the gateway Broker's current keep-alive setting for a particular cluster, you use the BrokerAdminClient.getClusterGatewayKeepAlive method. This method returns the gateway Broker's current keep-alive interval (in seconds) for a given cluster.

Pausing a Gateway Broker

To temporarily halt forwarding of events from one gateway Broker to another, you use the BrokerAdminClient.setClusterGatewayPause method. When you use this method, you specify the cluster whose events you want to hold and set the *pause* parameter to `true`.

Once you pause the forwarding of events to a particular cluster, the gateway Broker queues documents for that cluster in an event-forwarding queue. The Broker does not resume forwarding events to the remote cluster until event-forwarding is explicitly re-enabled.

Pausing Traffic in Both Directions

Be aware that the BrokerAdminClient.setClusterGatewayPause method halts the flow of traffic in one direction. That is, it stops the Broker to which the BrokerAdminClient client is connected from sending outbound messages to a specified cluster. It does not halt the flow of inbound traffic to the gateway Broker from the remote cluster. To halt both inbound and outbound between two gateway Brokers, you must execute BrokerAdminClient.setClusterGatewayPause on both Brokers.

Note that once you pause the forwarding queue for a cluster, it remains paused even if the gateway Broker is restarted or the Broker Admin client disconnects.

Pausing Traffic to and from Multiple Clusters

If a Broker serves as a gateway to multiple clusters, and you want to pause outbound traffic to all of the clusters, execute BrokerAdminClient.setClusterGatewayPause on the gateway Broker once for each remote cluster.

Similarly, if you want to halt inbound traffic to a gateway Broker from multiple remote clusters, execute BrokerAdminClient.setClusterGatewayPause on each remote gateway Broker.

Unpausing a Cluster Gateway Broker

To re-enable event forwarding, you re-execute BrokerAdminClient.setClusterGatewayPause and set the *pause* parameter to `false`.

Checking the Current Pause Setting

To check the state of event-forwarding queues on the gateway Broker, execute the BrokerAdminClient.getClusterStats method and examine the *gateway_pause* field in the Broker event that BrokerAdminClient.getClusterStats returns.

Static Gateway Forwarding

The static gateway forwarding feature ensures consistent document flow across a gateway between two clusters at all times. Documents defined by the gateway permissions list are allowed to flow over the gateway. Once static gateway forwarding is enabled, the documents are forwarded to the remote cluster at all times, regardless of any client subscriptions on the remote cluster.

Essentially, the static gateway forwarding feature force feeds documents to the remote cluster. When there are no subscribers on the remote cluster, forwarded documents will be dropped at the gateway peer of the remote cluster. Disabling static gateway forwarding resumes the normal behavior of subscription based gateway forwarding.

Note:
The static gateway forwarding feature is only supported on a webMethods Broker Version 6.1 Service Pack 4 and later. The Broker(s) on the other end of the gateway can be of any version. In addition, you cannot enable static gateway forwarding on a Broker that is either not in a cluster or does not have a gateway to the specified remote cluster.

To enable static gateway forwarding, use the BrokerAdminClient.setClusterStaticGatewayForwarding method and set *enable* to true.

To check the status of the static gateway subscription set on a remote cluster from the current Broker, execute the BrokerAdminClient.getClusterStaticGatewayForwarding method.

For more information about static gateway forwarding, see *Administering webMethods Broker*.

Refusing Document Type Updates Across a Gateway

By default, changes made to shared document types on one side of a gateway are propagated to a Broker cluster on the other side of the gateway. However, you can configure a cluster to refuse updates made on the other side of a gateway. This setting provides the following security measures:

- Lockdown of a local cluster. Unauthorized changes made in a neighboring cluster are not carried over to the local cluster.
- Protection against accidental changes made to shared document types in the neighboring cluster.

Configuring a cluster to refuse updates is done at the local end of a gateway. If you want the Broker clusters on both sides of a gateway to refuse updates, you must configure the setting from both sides.

Update Refusal Classes and Methods

You have the option of refusing all changes to document types made in the neighboring cluster, or specifying which document types cannot be changed.

- The following methods set and return whether a gateway refuses all updates made in a neighboring cluster: BrokerAdminClient.setClusterGatewayRefuseAllUpdates and BrokerAdminClient.getClusterGatewayRefuseAllUpdates.
- Use the BrokerAdminClient.getClusterGatewaySharedEventTypes to set and return the document types for which a gateway refuses updates.
- The BrokerSharedEventTypeRefuseUpdateInfo class stores information about how a particular event type is shared across a cluster gateway, and whether the event type itself or updates to the event type are refused.

Synchronizing Updates

If you turned on an update refusal setting for the local cluster and an update is made to a shared document type on the other side of the cluster gateway, all document flow to the local cluster stops. At this point, you need to synchronize the document types by using one of the methods described below:

- For update refusals set on the local side of a gateway, deliver a document of the type that was refused to the remote side of the gateway.
- Change the event type definition;
  - On the remote side of the gateway, revert the update back to the previous document type definition.
  - On the local side of the gateway, update the document type definition.
- Turn off the update refusal flag.

After you synchronize the updates, document flow to the local cluster will restart.

Note: Broker Server issues alerts for event flow suspension and resumption.

Setting Cluster Gateway Security

For information on cluster gateway security, see "Cluster Gateway Security Configurations" on page 61.

9 Managing Site Configurations

Overview

This chapter describes the webMethods Broker Java interfaces that allow you to load, store, import and export site configuration information for the following:

- Broker Servers
- Brokers
- Clients
- Client groups
- Event types
- Territories
- Territory gateways

These interfaces work in conjunction with the ActiveWorks Definition Language (ADL), described in *Administering webMethods Broker*.

Note:
See the *webMethods Broker Client Java API Programmer's Guide* for information on naming restrictions for Brokers, client groups, event types, and territories

Collection Information

The BrokerCompleteCollection class can hold information about an entire configuration of Broker Servers, Brokers, Broker clients, client groups, event types, and access control lists.

The following table shows the methods offered by the BrokerCompleteCollection class.

Method	Description
read readFile	Creates a BrokerCompleteCollection and initializes it from input data or from an input file containing ADL.
toString	Returns a string of ADL that describes the collection defined by this object.
write	Writes the contents of this collection, usually to a file, in ADL format.
writeVersionNumber	Writes just the version number of this object.

Server Information

You can use the BrokerCompleteServer class to hold the following information about a Broker Server:

- Access Control List

- Brokers contained in the server
- Server's description
- Server's host name
- Server's license key
- Server's log configuration
- Port number used
- Server's SSL configuration

The following table shows the methods offered by the BrokerCompleteServer class.

Method	Description
deepRefresh	Refreshes this object's data members, using the specified administrative Broker client.
deepRetrieve	Returns a BrokerCompleteServer object initialized using the specified administrative Broker client.
deepStore	Stores the information contained in this object into a Broker Server.
refresh	Refreshes this object's data members, using the specified administrative Broker client. Any information the Broker client does not have permission to access will not be set.
retrieve	Returns a BrokerCompleteServer object initialized using the specified administrative Broker client. Server information is returned, but not information on the server's Brokers.
store	Stores the information contained in this object into a Broker Server. Server information is stored, but not information the server's Brokers.
toString	Returns a string of ADL that describes the information contained in this object.
write	Writes the contents of this collection, usually to a file, in ADL format.

Broker Information

You can use the BrokerCompleteBroker class to hold information about a particular Broker. The following information is stored for the Broker:

- Broker's name
- Names of other Brokers in the Broker's territory
- Client groups
- Clients

- Broker's description
- Event types
- An indication if this is the default Broker
- Broker's territory name

The following table shows the methods offered by the BrokerCompleteBroker class.

Method	Description
deepRefresh	Refreshes this object's data members, using the specified administrative Broker client.
deepRetrieve	Returns a BrokerCompleteBroker object initialized using the specified administrative Broker client.
deepStore	Stores the information contained in this object into a Broker.
refresh	Refreshes this object's data members, using the specified administrative Broker client. Any information the Broker client does not have permission to access will not be set.
retrieve	Returns a BrokerCompleteBroker object initialized using the specified administrative Broker client.
store	Stores most of the information contained in this object into a Broker Server.
toString	Returns a string of ADL that describes the information contained in this object.
write	Writes the contents of this collection, usually to a file, in ADL format.

Client Information

You can use the BrokerCompleteClient class to hold information about a particular Broker client. The following information is stored for the Broker client:

- BrokerClientInfo for the Broker client
- Broker client's infoset
- Broker client's event subscription list
- An indication of whether or not the Broker client's event subscription list should be overwritten when the store method is invoked.

The following table shows the methods offered by the BrokerCompleteClient class.

Method	Description
refresh	Refreshes this object's data members, using the specified administrative Broker client. Any information the Broker client does not have permission to access will not be set.
retrieve	Returns a BrokerCompleteClient object initialized using the specified administrative Broker client.
store	Stores most of the information contained in this object into the Broker client.
toString	Returns a string of ADL that describes the information contained in this object.
write	Writes the contents of this collection, usually to a file, in ADL format.

Client Group Information

You can use the BrokerCompleteClientGroup class to hold information about a particular client group. The following information is stored for the client group:

- Client group's Access Control List
- Names of event types this group can publish
- Names of event types to which this group can subscribe
- ClientGroupInfo
- An indication if errors should be ignored when using the store or deepStore methods
- An indication if the can-publish and can-subscribe lists should be replaced when using the store or deepStore methods

The following table shows the methods offered by the BrokerCompleteClientGroup class.

Method	Description
refresh	Refreshes this object's data members, using the specified administrative Broker client. Any information the Broker client does not have permission to access will not be set.
retrieve	Returns a BrokerCompleteClientGroup object initialized using the specified administrative Broker client.
store	Stores the information contained in this object into the Broker.
toString	Returns a string of ADL that describes the information contained in this object.
write	Writes the contents of this collection, usually to a file, in ADL format.

Event Type Information

You can use the BrokerCompleteEventType class to hold information about a particular event type. The following information is stored for the event type:

- Event types infosets
- Event type's type definition
- An indication if the definition should be replaced when using the store or deepStore methods

The following table shows the methods offered by the BrokerCompleteEventType class.

Method	Description
refresh	Refreshes this object's data members, using the specified administrative Broker client. Any information the Broker client does not have permission to access will not be set.
retrieve	Returns a BrokerCompleteEventType object initialized using the specified administrative Broker client.
store	Stores the information contained in this object into the Broker.
toString	Returns a string of ADL that describes the information contained in this object.
write	Writes the contents of this collection, usually to a file, in ADL format.

Territory Information

You can use the BrokerCompleteTerritory class to hold information about a territory of Brokers. For more information about territories, see "Managing Broker Territories" on page 63.

The following information is stored for territory:

- Territory's Access Control List
- Names of the Brokers that belong to the territory
- BrokerTerritoryInfo, which contains a variety of information on the gateway, including its name, authentication type, and encryption level

The following table shows the methods offered by the BrokerCompleteTerritory class.

Method	Description
refresh	Refreshes this object's data members, using the specified administrative Broker client. Any information the Broker client does not have permission to access will not be set.

Method	Description
retrieve	Returns a BrokerCompleteTerritory object initialized using the specified administrative Broker client.
store	Stores the information contained in this object into the Broker.
toString	Returns a string of ADL that describes the information contained in this object.
write	Writes the contents of this collection, usually to a file, in ADL format.

Territory Gateway Information

You can use the BrokerCompleteTerritoryGateway class to hold information about a territory gateway. For more information about territory gateways, see "Managing Broker Territories" on page 63.

The following information is stored for a gateway:

- The gateway's Access Control List
- The names of all the shared event types that can be forwarded to the remote half of the territory gateway
- The names of all the shared event types that can be received from the remote half of the territory gateway
- BrokerTerritoryGatewayInfo, which contains the territory's name, authentication type, and encryption level

The following table shows the methods offered by the BrokerCompleteTerritoryGateway class.

Method	Description
refresh	Refreshes this object's data members, using the specified administrative Broker client. Any information the Broker client does not have permission to access will not be set.
retrieve	Returns a BrokerCompleteTerritory object initialized using the specified administrative Broker client.
store	Stores the information contained in this object into the Broker.
toString	Returns a string of ADL that describes the information contained in this object.
write	Writes the contents of this collection, usually to a file, in ADL format.

10 Monitoring Broker Activity

Overview

This chapter describes several ways that you can monitor the activity of a Broker. Reading this chapter will help you understand:

- The activity and trace events published by a Broker to which your client application can subscribe
- How you use the BrokerAdminClient and BrokerTraceEvent interfaces to obtain a Broker's activity traces
- How to use the BrokerLogConfig and BrokerLogEntry classes to configure the Broker's activity log
- How to access and manipulate Broker Server log entries

Activity and Trace Events

The webMethods Broker system provides two sets of events that provide information about changes in a Broker; activity events and trace events. Activity events describe changes in a Brokers clients, client groups, event types, and territory affiliation. Trace events provide information about the enqueuing, publishing, and forwarding of events performed by a Broker. Administrative tools or clients that dynamically adapt to changes in a Broker's configuration are two examples of applications that may want to subscribe to these events.

Each of the activity and trace event types described in this section are only published by a Broker if at least one client has subscribed to that event type. Any client with the necessary permissions may subscribe to these events and some of them can degrade the performance of your system, so you may want to consider restricting non-administrative applications from subscribing to them.

Note:
An activity or trace event can only be received by a Broker client connected to the Broker which published the event. Activity and trace events are local and will never be forwarded to another Broker. Furthermore, activity traces cannot be subscribed to using a wildcard subscription. See the *webMethods Broker Client Java API Programmer's Guide* for information on subscriptions and wildcards.

For information on configuring event subscription permissions, see *Administering webMethods Broker*.

Note:
Events are known as *documents* in *Administering webMethods Broker*.

Activity Events

Activity events provide information about the change in a Broker's configuration. These changes involve:

- Creation, disconnection, re-connection, destruction, client queue locking or unlocking for Queue Manipulation of a Broker client.

- Creation, modification, or destruction of a client group.
- Creation, modification, or destruction of an event type definition.
- A Broker joining or leaving a territory. For more information on territories, see "Managing Broker Territories" on page 63.

Client Changes

The event type Broker::Activity::ClientChange provides information about a change in a Broker's clients. This event consists of the event fields shown in the following table.

Field Name	Type	Description
`clientId`	`unicode_string`	The identifier of the Broker client whose state has changed.
`clientGroup`	`unicode_string`	The Broker client's group affiliation.
`appName`	`unicode_string`	The application name associated with the Broker client.
`stateFlags`	`int`	A bit mask with one or more of the following values: ■ 1-created ■ 2-destroyed ■ 4-connected ■ 8-disconnected ■ 16-client queue locked ■ 32-client queue unlocked
`user`	`unicode_string`	The user that made the change. This will only be set if the entity making the change used an SSL connection.
`authenticator`	`unicode_string`	The entity that authenticated the `user`. This will only be set if the entity making the change used an SSL connection.

Client Group Changes

The event type Broker::Activity::ClientGroupChange provides information about a change in a Broker's client groups. This event consists of the event fields shown in the following table.

Field Name	Type	Description
created	unicode_string[]	The list of client groups that were created.
changed	unicode_string[]	The list of client groups that were changed. Changes may include modifications to the can-publish list, can-subscribe list, or ACL.
destroyed	unicode_string[]	The list of client groups that were destroyed.
changeTime	date	The time the change occurred.
user	unicode_string	The user that made the change.
authenticator	unicode_string	The entity that authenticated the user.

Client Subscription Changes

The event type Broker::Activity::ClientSubscriptionChange provides information about a change in a Broker client's subscription. This event consists of the event fields shown in the following table.

Field Name	Type	Description
clientId	unicode_string	The identifier of the Broker client whose state has changed.
clientGroup	unicode_string	The Broker client's group affiliation.
appName	unicode_string	The application name associated with the Broker client.
eventTypeName	unicode_string	The name of the event type.
filterExpr	unicode_string	The filter used in the subscription.
action	int	A bit mask with one or more of the following values: ■ 1-created ■ 2-cancelled
user	unicode_string	The user that made the change. This will only be set if the entity making the change used an SSL connection.
authenticator	unicode_string	The entity that authenticated the user. This will only be set if the entity making the change used an SSL connection.

Event Type Changes

The event type Broker::Activity::EventTypeChange provides information about a change in a Broker's event type definitions. This event consists of the event fields shown in the following table.

Field Name	Type	Description
`created`	`unicode_string[]`	The list of event types that were created.
`defChanged`	`unicode_string[]`	The list of event types whose definitions were changed.
`infoChanged`	`unicode_string[]`	The list of event types whose infosets were modified.
`destroyed`	`unicode_string[]`	The list of event types that were destroyed.
`changeTime`	`date`	The time the change occurred.
`user`	`unicode_string`	The user that made the change.
`authenticator`	`unicode_string`	The entity that authenticated the `user`.

Remote Subscription Changes

The event type Broker::Activity::RemoteSubscriptionChange provides information about a change in a remote Broker's subscription. This event consists of the event fields shown in the following table.

Field Name	Type	Description
`territoryName`	`unicode_string`	The territory name of the remote Broker.
`brokerName`	`unicode_string`	The name of the remote Broker for which the subscription is changed.
`hostName`	`unicode_string`	The host name where the remote Broker is running.
`eventTypeName`	`unicode_string`	The name of the event type.
`filterExpr`	`unicode_string`	The filter used in the subscription.
`action`	`int`	A bit mask with one or more of the following values: ■ 1-created ■ 2-cancelled
`user`	`unicode_string`	The user that made the change. This will only be set if the entity

Field Name	Type	Description
		making the change used an SSL connection.
`authenticator`	`unicode_string`	The entity that authenticated the `user`. This will only be set if the entity making the change used an SSL connection.

Territory Changes

The event type Broker::Activity::TerritoryChange provides information about a change in a Broker's territory affiliation. This event will be published by a Broker when another Broker joins or leaves its territory. The event consists of the event fields shown in the following table.

Field Name	Type	Description
`territoryName`	`unicode_string`	The name of the Broker's territory.
`brokerName`	`unicode_string`	The name of the Broker.
`hostName`	`unicode_string`	The name of the Broker's host.
`changeFlags`	`int`	A bit mask with one or more of the following values: ■ 1-jointed ■ 2-left ■ 4-local Broker
`user`	`unicode_string`	The user that made the change.
`authenticator`	`unicode_string`	The entity that authenticated the `user`.

Trace Events

Trace events provide information about events that are passing through the system and are useful in monitoring and evaluating event traffic. Trace events themselves never cause other trace events to be generated, yet they may still add considerable overhead to your system. The same information is available through the getActivityTraces method, which will not degrade the performance of your system. Each trace event type is published only if at least one client has subscribed to that event type.

Note:
Traces cannot be subscribed to using a wildcard subscription. See the *webMethods Broker Client Java API Programmer's Guide* for information on subscriptions and wildcards.

Dropped Events

The event type Broker::Trace::Drop is published whenever an event was discarded by the Broker because it was delivered to a Broker client which did not have permission to receive the event. This event consists of the event fields shown in the following table.

Field Name	Type	Description
clientId	unicode_string	The Broker client's identifier.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.
pubId	unicode_string	The identifier of the Broker client that delivered the event.
destId	unicode_string	The identifier of the intended recipient Broker client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Remotely Dropped Events

The event type Broker::Trace::DropRemote is published whenever an event was discarded by the Broker while forwarding to other territory Brokers because the event was too large to be handled by the territory Broker. This occurs in a territory consisting of 5.0 and pre 5.0 Brokers while forwarding an event of size greater than the maximum transaction size from 5.0 Broker to pre 5.0 Brokers. This event consists of the event fields shown in following table.

Field Name	Type	Description
brokerName	unicode_string	The name of the remote Broker for which the event was enqueued.
brokerHost	unicode_string	The host name where the remote Broker is running.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.
pubId	unicode_string	The identifier of the Broker client that delivered the event.
destId	unicode_string	The identifier of the intended recipient Broker client.

Field Name	Type	Description
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Enqueued Events

The event type Broker::Trace::Enqueue is published whenever the Broker places an event in a Broker client's queue. This event consists of the event fields shown in the following table.

Field Name	Type	Description
clientId	unicode_string	The Broker client's identifier.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.
pubId	unicode_string	The identifier of the Broker client that delivered the event.
destId	unicode_string	The identifier of the intended recipient Broker client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Remotely Enqueued Events

The event type Broker::Trace::EnqueueRemote is published whenever the Broker puts an event in a remote Broker's queue as part of Broker-to-Broker communication. This event consists of the event fields shown in the following table.

Field Name	Type	Description
brokerName	unicode_string	The name of the remote Broker for which the event was enqueued.
brokerHost	unicode_string	The host name where the remote Broker is running.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.

Field Name	Type	Description
pubId	unicode_string	The identifier of the Broker client that delivered the event.
destId	unicode_string	The identifier of the intended recipient Broker client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Published Events

The event type Broker::Trace::Publish is published whenever the Broker publishes or delivers an event. This event consists of the event fields shown in the following table.

Field Name	Type	Description
clientId	unicode_string	The Broker client's identifier.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.
pubId	unicode_string	The identifier of the Broker client that delivered the event.
destId	unicode_string	The identifier of the intended recipient Broker client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Remotely Published Event

The event type Broker::Trace::PublishRemote is published whenever the Broker receives a published event from a remote Broker. This event consists of the event fields shown in the following table.

Field Name	Type	Description
brokerName	unicode_string	The name of the remote Broker for which the event was enqueued.

Field Name	Type	Description
brokerHost	unicode_string	The host name where the remote Broker is running.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.
pubId	unicode_string	The identifier of the Broker client that delivered the event.
destId	unicode_string	The identifier of the intended recipient Broker client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Received Events

The event type Broker::Trace::Receive is published whenever a Broker client acknowledges an event that it has retrieved from its client queue. This event consists of the event fields shown in the following table.

Field Name	Type	Description
clientId	unicode_string	The Broker client's identifier.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.
pubId	unicode_string	The identifier of the Broker client that published the event.
destId	unicode_string	The identifier of the intended recipient Broker client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Remotely Received Event

The event type Broker::Trace::ReceiveRemote is published whenever a remote Broker acknowledges the receipt of a forwarded event. This event consists of the event fields shown in the following table.

Field Name	Type	Description
brokerName	unicode_string	The name of the remote Broker.
brokerHost	unicode_string	The host name where the remote Broker is running.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.
pubId	unicode_string	The identifier of the Broker client that published the event.
destId	unicode_string	The identifier of the intended recipient Broker client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Peek Events

The event type Broker::Trace::Peek is published whenever an event is peeked administratively from the Broker client's queue. This event consists of the event fields shown in the following table.

Field Name	Type	Description
clientId	unicode_string	The Broker client's identifier.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.
lockingId	unicode_string	The identifier of the Broker client that is holding a queue lock on this client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Insert Events

The event type Broker::Trace::Insert is published whenever an event is inserted administratively into the Broker client's queue. This event consists of the event fields shown in the following table.

Field Name	Type	Description
clientId	unicode_string	The Broker client's identifier.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.
lockingId	unicode_string	The identifier of the Broker client that is holding a queue lock on this client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	Int	The size of the event in bytes.

Delete Events

The event type Broker::Trace::Delete is published whenever an event is deleted administratively from the Broker client's queue. This event consists of the event fields shown in the following table.

Field Name	Type	Description
clientId	unicode_string	The Broker client's identifier.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.
lockingId	unicode_string	The identifier of the Broker client that is holding a queue lock on this client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Modify Events

The event type Broker::Trace::Modify is published whenever an event is modified administratively from the Broker client's queue. This event consists of the event fields shown in the following table.

Field Name	Type	Description
clientId	unicode_string	The Broker client's identifier.
eventTypeName	unicode_string	The name of the event type.
eventId	long	The event identifier from the event.

Field Name	Type	Description
seqnNumber	long	The sequence number of the event.
oldSeqnNumber	long	The sequence number of the deleted event.
lockingId	unicode_string	The identifier of the Broker client that is holding a queue lock on this client.
trackId	unicode_string	The tracking identifier from the event.
tag	int	The event's tag field.
eventByteSize	int	The size of the event in bytes.

Activity Traces

Each Broker maintains its own set of activity traces. Activity traces require very little overhead and will not degrade the performance of your system. However, activity traces can be lost if the Broker is very busy or if your client does not retrieve them quickly enough.

A BrokerTraceEvent is created by a Broker whenever one of the following occurs:

- A Broker client or remote Broker is created or destroyed.
- A Broker client or remote Broker publishes or delivers an event.
- A Broker client or remote Broker retrieves an event from its event queue.
- A Broker adds an event to a client's or remote Broker's event queue.
- An event was dropped from a Broker client's queue because the client did not have permission to receive the event.

Understanding the BrokerTraceEvent

The BrokerTraceEvent class represents an entry in a Broker's activity trace. Some of the information that may be contained in a trace event includes:

- A key indicating the trace event's type
- The name of the client application
- The name of the Broker Server
- The name of the Broker host
- The application's client group and client identifier, if applicable
- The event type name
- The event's `_env.tag` field, destination ID, and receiving client ID

- The client identifier of the destination client, if applicable

Obtaining Activity Traces

You can use the BrokerAdminClient.getActivityTraces method to obtain an array of BrokerTraceEvent objects. You must supply the sequence number of the first trace event to be retrieved and the number of milliseconds to wait if no trace information is available.

The first time you invoke this method, the sequence number should be set to zero. On subsequent invocations of this method, you should set the sequence number one plus the value of the last trace event's sequence number.

Broker Logging

Each Broker maintains a log containing three levels of failure messages, which may be configured individually.

Level	Description
Alert	Identifies critical messages.
Warning	Identifies important messages.
Informational	Identifies informational messages.

The Broker's log may use any of the following logging mechanisms:

- Unix syslog facility
- Windows NT event log
- SNMP alerts

Understanding the BrokerLogConfig

You can use the BrokerServerClient.getLogConfig method to obtain the logging configuration for a Broker Server. The BrokerServerClient.setLogConfig method allows you to set the logging configuration for a Broker Server.

The BrokerLogConfig class represents the Broker Server's logging configuration and provides methods that allow you to:

- Obtain log configuration entries by topic.
- Obtain the entire contents of the log configuration.
- Remove a single entry from the log configuration or remove all entries.
- Remove log configuration entries by topic

The following table shows the methods offered by the BrokerLogConfig class.

Method	Description
clearLogOutput clearLogOutputs	Clears one or more entries from a specified log output.
clearLogTopic clearLogTopics	Clears one or more entries from a specified log topic.
getLogOutput clearLogOutputs	Returns one or more BrokerLogEntry objects which represent the state of the specified log output or outputs.
getLogTopic getLogTopics	Returns one or more BrokerLogEntry objects which represent the state of the specified log topic or topics.
setLogOutput	Sets the value of a log output's state.
setLogTopic	Sets the value of a log topic's state.
toString	Returns a string containing the log configuration in a human-readable format.

Understanding the BrokerLogEntry

The BrokerLogConfigEntry class represents the settings for an instance of a log output or a log topic. It contains the following information:

- The type of log output or topic being represented.
- An indication of whether or not the output or topic is enabled.
- An optional string containing additional information.

Broker Server Logs

Each Broker Server maintains a log of significant events that have occurred. The format of each entry described by the BrokerServerLogEntry object is shown in the following table.

Field Name	Type	Description
`time_stamp`	`BrokerDate`	The date and time that the entry was created.
`entry_type`	`short`	The entry's type. See the next table, below, for a list of possible types.
`entry_mesg_id`	`short`	The message identifier.
`entry_msg_text`	`char *`	The event identifier from the event.

Each Broker Server log entry is assigned a type as defined in the following table.

Constant	Description
SERVER_LOG_MESSAGE_UNKNOWN	The entry has an undefined type.
SERVER_LOG_MESSAGE_INFO	The entry contains an informational message.
SERVER_LOG_MESSAGE_WARNING	The entry contains a warning message.
SERVER_LOG_MESSAGE_ALERT	The entry contains an alert message.

Obtaining Log Entries

You can use the BrokerServerClient.getServerLogEntries method to obtain a desired set of entries from the log of the Broker Server to which your client is connected. This method lets you specify the date of the first entry in which you are interested and the total number of entries you wish to retrieve.

Note:
Attempting to obtain all server log entries may create performance problems and may use up significant amounts of memory.

The BrokerServerLogEntry.toString method allows you to convert a BrokerServerLogEntry to a character string.

Obtaining Log Information

The BrokerServerLogInfo object describes basic information about a particular server's log. The following table describes the contents of the BrokerServerLogInfo object.

Field Name	Type	Description
num_entries	int	The number of entries contained in the log.
first_entry	BrokerDate	The date and time of the earliest entry contained in the log.
last_entry	BrokerDate	The date and time of the most recent entry contained in the log.

You can use the BrokerServerClient.getServerLogStatus method to obtain the status of the log of the Broker Server to which your client is connected.

Deleting Log Entries

You can use the BrokerServerClient.pruneServerLog method to delete selected entries from the log of the Broker Server to which your client is connected. All entries with dates equal to or older than the date you specify will be deleted.

11 Using Queue Browsers

Overview

This chapter describes several ways by which you manage client queues. Reading this chapter will help you understand:

- How to use `BrokerQueueBrowser`, `BrokerLockedQueueBrowser`, `BrokerQueueBrowserInfo`, and `BrokerClient` and `BrokerAdminClient` classes to examine a queue's content.
- How to rearrange the contents of the client queue by adding, deleting, and replacing documents.
- How to rearrange the contents of the forwarding queue for a remote Broker by adding, deleting, and replacing documents.

> **Note:**Broker events are called Broker *documents types* in Software AG Designer and other webMethods components.

Client Queue Browser

webMethods Broker Java API 5.x and 6.1 provided the queue manipulation API with a limited ability to manipulate the contents of a client queue. In webMethods Broker Java API 6.5 and later, the queue manipulation API is replaced with the client queue browser API, which supports queue management in a much more robust manner. The following are the key differences between the new queue browser and the former queue manipulation feature.

The client queue browser supports three distinct modes of operation:

1. A client can inspect its own queue contents without requiring any administrative privileges.
2. An administrative client can inspect a client queue without acquiring the client's queue lock.
3. An administrative client can acquire a queue lock and perform an "in-place" replacement of documents in the locked queue, in addition to the usual insert, delete and browse operations.

You can perform four operations with the queue browser. Note that you can only perform the last three operations on a locked queue.

1. The browse operation to return queued documents, including the documents that have been retrieved but are not yet acknowledged (i.e., "unacknowledged") documents. You can apply filtering on browse operations.
2. The insert operation to add documents to either the head or tail of a locked queue.
3. The delete operation to delete documents based on their receipt sequence numbers.
4. The modify operation to replace documents "in-place" based on their receipt sequence numbers.

Using Client Queue Browser

Client queue browser works in three different modes.

1. **Client queue browser.** A client can open a queue browser on its own queue and browse documents without requiring any administrative privileges. A client queue browser requires

a dedicated client session and allows browse operations on the queue from any queue position and exposes all documents in the queue.

2. **Administrative Queue Browser - Unlocked Mode.** An administrative queue browser allows an administrative client to access the contents of a client queue owned by a different client at runtime. In the unlocked browse mode the client queue is not locked, meaning that the queue content could change between queue browser operations. In this unlocked mode, queue browsing is the only operation the administrative client can perform. The existence of an administrative queue browser in an unlocked mode does not impact the normal functioning of the client queue.
3. **Administrative Queue Browser - Locked Mode.** An administrative queue browser allows an administrative client to access the contents of a client queue owned by a different client at runtime. In this mode, the client queue is locked for the lifetime of the queue browser. You can browse queue contents, delete documents from queue, insert new documents into the queue, and modify documents in the queue.

In order to invoke any queue browser operation on a client queue, you must first create a client queue browser using one of the following methods:

- BrokerClient.createClientQueueBrowser()

  Broker client creates a queue browser to browse its own queue.

- BrokerAdminClient.createClientQueueBrowser()

  Broker administrative client creates a queue browser to browse another client's queue

- BrokerAdminClient.createLockedClientQueueBrowser()

  Broker administrative client creates a queue browser to manipulate the contents of another client's queue.

These functions will return a queue browser that can be used to invoke various queue management functions on the client queue.

Note:
Pre-6.5 Brokers do not support the queue browser API. You will receive an error if your client attempts to open a queue browser on pre-6.5 versions of Broker clients.

The BrokerClientQueueBrowserInfo object represents information of a client queue browser with data members described in the following table.

Data Member	Description
`String clientId`	This member contains the ID of the client on which the queue browser is currently open. For information on client IDs, see "Managing Broker Clients" on page 25.
`String browserId`	This member contains the ID of the client that created the queue browser. For information on client IDs, see "Managing Broker Clients" on page 25.

Data Member	Description
`int browserSessionId`	This member contains the session ID of the client that created the queue browser.
`boolean locked`	The value is set to true if the client queue is locked by this queue browser; otherwise, false.
`int createTime`	This member contains the date and time when the queue browser was created.

Forwarding Queue Browser

webMethods Broker Java API 8.2 and later provides an API to view the contents of a forwarding queue for a remote Broker. The forwarding queue browser supports two distinct modes of operation:

1. An administrative client can inspect a forwarding queue without acquiring the client's queue lock.
2. An administrative client can acquire a queue lock and perform browse, delete, insert, and modify operations.

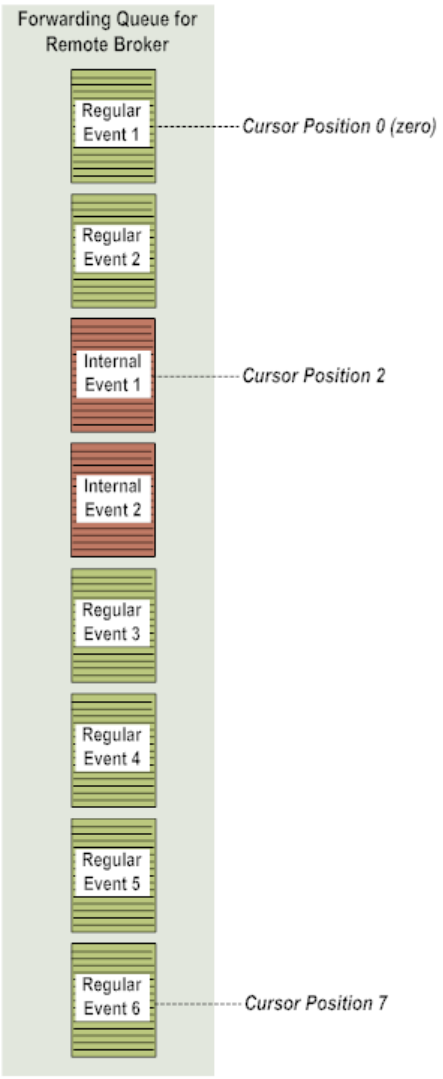
You can perform four operations with the queue browser. Note that you can only perform the last three operations on a locked queue.

1. The browse operation to return queued documents, including the documents that have been retrieved but are not yet acknowledged (i.e., "unacknowledged") documents. You can apply filtering on browse operations.
2. The insert operation to add documents to the tail of a locked queue.

   Note:
   You cannot insert documents to the head of a forwarding queue for a remote Broker.

3. The delete operation to delete documents based on their receipt sequence numbers.
4. The modify operation to replace documents "in-place" based on their receipt sequence numbers.

In addition to regular documents, the forwarding queue for a remote Broker might also contain internal events that Broker uses to perform tasks such as territory synchronization, and subscription updates. Internal events are produced only by Brokers and consumed only by peer Brokers in a territory, cluster, or a gateway Broker. Regular clients cannot process internal events because internal events are not associated with any Broker event type. The queue browser skips all internal events while returning documents from a forwarding queue as explained in the following diagram.



In this illustration, if you set the cursor position to "2" and request six events, the queue browser returns only four events namely, Regular Event 3, Regular Event 4, Regular Event 5, and Regular Event 6. This behavior occurs because the events at cursor position 2 and cursor position 3 are internal events. The queue browser skips these events and returns the remaining events from the forwarding queue.

You can delete regular events from the forwarding queue for a remote Broker, but you cannot delete internal events. An internal event might contain metadata changes. So deleting an internal event would mean that the regular events that come next in the queue might not be consumed successfully by the subscribers due to metadata mismatch.

Using Forwarding Queue Browser

Forwarding queue browser works in two modes.

1. **Administrative Queue Browser - Unlocked Mode.** An administrative queue browser allows an administrative client to access the contents of a forwarding queue for a remote Broker at

runtime. In the unlocked browse mode the forwarding queue is not locked, meaning that the queue content could change between queue browser operations. In this unlocked mode, queue browsing is the only operation the administrative client can perform. The existence of an administrative queue browser in an unlocked mode does not impact the normal functioning of the forwarding queue.

2. **Administrative Queue Browser - Locked Mode.** An administrative queue browser allows an administrative client to access the contents of a forwarding queue for a remote Broker at runtime. In this mode, the forwarding queue is locked for the lifetime of the queue browser. You can browse queue contents, delete documents from queue, insert new documents into the queue, and modify documents in the queue.

To invoke a queue browser operation on the forwarding queue, the Broker administrative client must first create a forwarding queue browser using the BrokerAdminClient.createForwardQueueBrowser()method.

To invoke a queue browser operation on the forwarding queue, the Broker administrative client must first create a forwarding queue browser using one of the following methods:

- BrokerAdminClient.createForwardQueueBrowser()

  Broker administrative client creates a queue browser to browse forwarding queue.

- BrokerAdminClient.createLockedForwardQueueBrowser()

  Broker administrative client creates a queue browser to manipulate the contents of a forwarding queue.

This method will return a queue browser that you use to invoke various queue management functions on the queue.

Note: Brokers earlier than version 8.1 do not support the queue browser APIs.

The BrokerQueueBrowserInfo object represents information of a queue browser with data members described in the following table.

Data Member	Description
`String clientId`	This member contains the ID of the client or remote Broker on which the queue browser is currently open.
`String browserId`	This member contains the ID of the client that created the queue browser. For information on client IDs, see "Using Queue Browsers" on page 111.
`int browserSessionId`	This member contains the session ID of the client that created the queue browser.
`boolean locked`	The value is set to true if the client queue is locked by this queue browser; otherwise, false.
`int createTime`	This member contains the date and time when the queue browser was created.

Creating and Closing Client Queue Browsers

Use the following methods to create and close client queue browsers in each mode.

For a Client Queue Browser

- Create a queue browser using the BrokerClient.createClientQueueBrowser() method. This method creates a queue browser on the client's queue.

  Note:
  This mode of operation does not require a client queue to be locked; therefore the client queue content can undergo changes while the queue browser is open.

- Close the queue browser using the BrokerClientQueueBrowser.closeClientQueueBrowser() method.

The following example illustrates how to create a client queue browser on client's own queue:

```
BrokerClient client = null;
BrokerClientQueueBrowser browser = null;

//Create a Broker client
try {
  client = new BrokerClient(broker_host,
             broker_name,
                           client_id,
                           client_group,
                           "QueueBrowser - SELF",
                           null);
} catch(BrokerException ex) {
  System.out.println("Failed to create client.");
  return;
}
. . .
//Create a queue browser
try {
  browser = client.createClientQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to create client queue browser.");
  return;
}
. . .
//Various queue browser operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close client queue browser.");
  return;
}
```

For an Administrative Queue Browser With No Lock

- Create a queue browser using the BrokerAdminClient.createClientQueueBrowser() method. This creates an administrative queue browser on the client's queue.

  Note:
  This mode of operation does not require a client queue to be locked; therefore the client queue content can undergo changes while the queue browser is open.

- Close the queue browser using BrokerClientQueueBrowser.closeQueueBrowser() method.

The following example illustrates how to create an administrative queue browser on a client queue (UNLOCKED mode):

```
BrokerClient client = null;
BrokerAdminClient admin = null;
BrokerClientQueueBrowser browser = null;

//Create a Broker client
try {
  client = new BrokerClient(broker_host,
             broker_name,
                            client_id,
                            client_group,
                            "QueueBrowser ",
                            null);
} catch(BrokerException ex) {
  System.out.println("Failed to create client.");
  return;
}
//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                 broker_name,
                                 null,
                                 "admin",
                                 "QueueBrowser-UNLOCKED",
                                 null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .
//Create a queue browser in unlocked mode
try {
  browser = admin.createClientQueueBrowser(client.getId());
} catch(BrokerException ex) {
  System.out.println("Failed to create client queue browser.");
  return;
}
. . .
//Various browse operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
```

```
  System.out.println("Failed to close client queue browser.");
  return;
}
```

For a Administrative Queue Browser With a Lock

- Create a queue browser using the BrokerAdminClient.createLockedClientQueueBrowser() method. This method creates a locked administrative queue browser on the client's queue.
- Close the queue browser using the BrokerClientQueueBrowser.closeQueueBrowser() method.

The following example illustrates how to create an administrative queue browser on a client queue (LOCKED mode):

```
BrokerClient client = null;
BrokerAdminClient admin = null;
BrokerLockedClientQueueBrowser browser = null;

//Create a Broker client
try {
  client = new BrokerClient(broker_host,
             broker_name,
                           client_id,
                           client_group,
                           "QueueBrowser",
                           null);
} catch(BrokerException ex) {
  System.out.println("Failed to create client.");
  return;
}
//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                 broker_name,
                                 null,
                                 "admin",
                                 "QueueBrowser-LOCKED",
                                 null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .
//Create a locked queue browser
try {
  browser = admin.createLockedClientQueueBrowser(client.getId());
} catch(BrokerException ex) {
  System.out.println("Failed to create locked client queue browser.");
  return;
}
. . .
//Various queue browser operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close client queue browser.");
```

```
  return;
}
```

Creating and Closing Forwarding Queue Browsers

Use the following methods to create and close forwarding queue browsers in unlocked mode.

For an Administrative Queue Browser With No Lock

- Create a queue browser using the BrokerAdminClient.createForwardQueueBrowser() method. This creates an administrative queue browser on the forwarding queue for the remote Broker.

  Note:
  This mode of operation does not require a queue to be locked; therefore the queue content can undergo changes while the queue browser is open.

- Close the queue browser using BrokerClientQueueBrowser.closeQueueBrowser() method.

The following example illustrates an administrative queue browser on a forwarding queue (UNLOCKED mode):

```
BrokerAdminClient admin = null;
BrokerQueueBrowser browser = null;
   //Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                               null,
                               "admin",
                               "ForwardQueueBrowser-UNLOCKED",
                               null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .
//Create a queue browser in unlocked mode
try {
  browser = admin.createForwardQueueBrowser(remote_broker_name);
} catch(BrokerException ex) {
  System.out.println("Failed to create forward queue browser.");
  return;
}
. . .
//Various browse operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close forward queue browser.");
  return;
}
```

Acquiring a Queue Lock

When you create a BrokerLockedQueueBrowser the target queue is automatically locked. Information on the queue browser can be obtained with getBrowserInfo call. One of the members of the BrokerQueueBrowserInfo indicates whether the queue is locked or not ('locked'). If the target queue is already locked by another admin client, an exception will be thrown. A queue cannot be locked by more than one admin client at any given time.

After the target queue is locked, the target queue maintains a "QueueLocked" state, which prevents certain operation from changing the queue state. For example:

- A getEvent(s) call on a locked target client queue will timeout.
- Events from a locked forwarding queue will not be delivered to the target remote Broker.
- A clear client queue operation will be rejected with an error message notifying the client queue locked status.
- A client or session disconnect (via the Broker Administrator tool) or client destroy commands will be rejected. An error message will be returned, notifying that the queue is locked.
- A command such as "leave territory" that destroys a forwarding queue for a remote Broker is rejected. An error message will be returned, notifying that the queue is locked.

All acknowledgements, both implicit and explicit, will be processed. The delete and modify queue browser operations only consider the deliverable documents for operation.

Releasing a Queue Lock

When you close a BrokerLockedQueueBrowser the target queue is automatically unlocked. If the administrative client that is holding the queue lock for some reason disconnects or is deleted, the queue lock will be automatically released. Once the queue is unlocked, all the requests from the client are honored.

Queue Position

The browse operation of a queue browser is a position-based operation. A queue browser internally maintains a cursor that points to a valid queue position anywhere between the head to the tail of the queue (between 0 to "queue length minus one"). Upon creation of a queue browser, the cursor is set to point to the head of the queue. After the browse operation, the cursor then points to the next document position for the following browse operation.

For example, if the cursor points to a queue position of 200 and the browse operation is for 10 documents, the queue browser returns 10 documents from the current position of 200 to position 209. After the browse operation, the cursor is now pointing at position 210 in the queue for the next browse operation.

You can alter queue position at any point of time using the setPosition method. Any subsequent browse operation will start browsing documents from that queue position.

The following example illustrates how to set the cursor position to queue position 200:

```
BrokerQueueBrowser browser = null;

. . .
//Create a queue browser
...
//Set queue position
try {
  browser.setPosition( 200); //Set at queue position 200
} catch(BrokerException ex) {
  System.out.println("Failed to set queue position.");
  return;
}
. . .
//Various queue browser operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close queue browser.");
  return;
}
```

Filters on a Queue Browser

You can set and reset filters on a queue browser at any time during the lifetime of the queue browser. A browse operation will return only the documents that satisfy the filters, if any are set on the browser.

For example, the queue *myQ* has 10 documents and 3 contain the string "Hello webMethods!". When a filter for the string "Hello" is applied to *myQ*, all subsequent browse operations will return only on the three documents containing "Hello webMethods!".

Note that queue browser filters are applicable only to the browse operation and not other queue browser operations, such as insert or delete operations. One or more filters can be set at any time during the queue browsing session. Use BrokerFilterDescriptor object to describe filters.

The following APIs are provided for setting filters on a queue browser.

Setting a Single Filter on Queue Browser

Use the setFilter method to set up a single filter for queue browse operations. If the queue browser already has one or more filters, this new filter is added to the list. If a filter is already present for a document type, the existing filter will be overwritten with the new filter. When one or more filters are set for the queue browse operations, only documents matching those filters are processed.

The following example illustrates how to set a single filter:

```
BrokerAdminClient admin = null;
BrokerQueueBrowser browser = null;
BrokerFilterDescriptor filter =
   new BrokerFilterDescriptor(""API::Test", "test_int < 10");
. . .
//Create a Broker admin client
try {
```

```
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                               null,
                               "admin",
                               "QueueBrowser-UNLOCKED",
                               null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}

//Create a queue browser
. . .

//Set single filter
try {
  browser.setFilter(filter);
} catch(BrokerException ex) {
  System.out.println("Failed to set filter.");
}
. . .

//Various browse operations
. . .

//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close queue browser.");
  return;
}
```

Setting Multiple Filters on Queue Browser

Use the setFilters method to set up multiple filters for queue browse operations. Note that queue browser filters are applicable only to the browse operation and not other queue browser operations, such as insert or delete operations.

If the queue browser already has one or more filters, the new filters are added to the list. If a filter is already present for a document type, the existing filter will be overwritten with the new filter. When filters are set for the queue browse operations, only documents matching those filters are processed.

The following example illustrates how to set multiple filters:

```
BrokerAdminClient admin = null;
BrokerQueueBrowser browser = null;
BrokerFilterDescriptor[] filters = new BrokerFilterDescriptor[3];

filters[0] = new BrokerFilterDescriptor(""API::Test", "test_int < 10");
filters[1] = new BrokerFilterDescriptor(""API::Test1",
        "my_string.contains("Hello"));
filters[2] = new BrokerFilterDescriptor(""API::Test2", "test_byte%2 == 0"));

//Create a Broker admin client
try {
```

```
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                                null,
                                "admin",
                                "QueueBrowser-UNLOCKED",
                                null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .
//Create a queue browser
. . .
//Set multiple filters
try {
  browser.setFilters(filters);
} catch(BrokerException ex) {
  System.out.println("Failed to set multiple filters.");
}
. . .
//Various browse operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close queue browser.");
  return;
}
```

Resetting Filters on Queue Browser

Use the resetFilters method to reset the filter list on the current queue browser. The filter list on the queue browser will be emptied out. Once the filters are reset, a browse request will return all documents with no filtering applied.

The following example illustrates how to reset filters:

```
BrokerAdminClient admin = null;
BrokerQueueBrowser browser = null;
//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                                null,
                                "admin",
                                "QueueBrowser-UNLOCKED",
                                null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .

//Create a queue browser in locked or unlocked mode
...
//Reset filters
try {
```

```
  browser.resetFilters();
} catch(BrokerException ex) {
  System.out.println("Failed to reset filters.");
}
. . .

//Various browse operations
. . .

//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close queue browser.");
  return;
}
```

Viewing Currently Set Filters on Queue Browser

Use the getFilters method to view the current filter set on the client queue lock. The filters are returned as BrokerFilterDescriptor objects.

The following example illustrates how to view set filters:

```
BrokerAdminClient admin = null;
BrokerQueueBrowser browser = null;
BrokerFilterDescriptor[] inFilters = null;
BrokerFilterDescriptor[] outFilters = null;
//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                              null,
                              "admin",
                              "QueueBrowser-UNLOCKED",
                              null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .

//Create a queue browser in locked or unlocked mode
...

// Set filters on browser
 browser.setFilters(inFilters);

. . .
//Various browse operations
. . .
//Get filters applied on browser
try {
  outFilters = browser.getFilters();
} catch(BrokerException ex) {
  System.out.println("Failed to get filters.");
}
. . .
```

```
//Various browse operations
. . .

//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close client queue browser.");
  return;
}
```

Viewing Queue Content

Viewing a Client's Own Queue Content

For a client to create a queue browser on its own queue and browse documents, follow these steps.

1. Create a queue browser using the BrokerClient.createClientQueueBrowser() method. This will create a queue browser on the client's queue.

   Note:
   This mode of operation does not require a client queue to be locked; therefore the client queue content can undergo changes while the queue browser is open.

2. Set a filter or filters, if needed. Note that once a filter is set for the queue browser, all browse operations will only operate on documents that pass through the filters. See "Filters on a Queue Browser" on page 122 for instructions.

3. Use browseEvents to view the content of the queue. See "BrokerQueueBrowser and BrokerClientQueueBrowser" on page 139 for instructions.

The following example illustrates how to examine one's own queue content:

```
BrokerClient client = null;
BrokerAdminClient admin = null;
BrokerQueueBrowser browser = null;
BrokerEvent[] events = null;
int max_events = 100; //Maximum number of documents to be browsed
int msecs = 30000;   //Timeout for the browse operation

//Create a Broker client
try {
  client = new BrokerClient(broker_host,
             broker_name,
                           client_id,
                           client_group,
                           "QueueBrowser - SELF",
                           null);
} catch(BrokerException ex) {
  System.out.println("Failed to create client.");
  return;
}
. . .
//Create a queue browser
```

```
try {
  browser = client.createClientQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to create client queue browser.");
  return;
}
. . .
//Browse operation
try {
  events = browser.browseEvents( max_events, msecs);
} catch(BrokerException ex) {
  System.out.println("Failed to complete browse operation.");
  return;
}
. . .
//Various browse operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close  queue browser.");
  return;
}
```

Administratively Viewing Contents of an Unlocked Client Queue

For an administrative client to examine client queue content in an unlocked mode, follow these steps.

1. Create a queue browser in an unlocked mode using the BrokerAdminClient.createClientQueueBrowser(String client_id) method. This will create a queue browser on the specified client's queue.

   Note:
   This mode of operation does not require a client queue to be locked; therefore the client queue content can undergo changes while the queue browser is open.

2. Set a filter or filters, if needed. It is important to note that once a filter is set for the queue browser, all browse operations will consider only documents that pass the filters. See "Filters on a Queue Browser" on page 122 for instructions.

3. Use browseEvents to view the content of the queue. See "BrokerLockedQueueBrowser and BrokerLockedClientQueueBrowser" on page 141 for instructions.

The following example illustrates how to examine a client queue in unlocked mode:

```
BrokerClient client = null;
BrokerAdminClient admin = null;
BrokerQueueBrowser browser = null;
BrokerEvent[] events = null;
int max_events = 100; //Maximum number of documents to be browsed
int msecs = 30000;   //Timeout for the browse operation

//Create a Broker client
try {
```

```
  client = new BrokerClient(broker_host,
             broker_name,
                            client_id,
                            client_group,
                            "QueueBrowser ",
                            null);
} catch(BrokerException ex) {
  System.out.println("Failed to create client.");
  return;
}
//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                 broker_name,
                                 null,
                                 "admin",
                                 "QueueBrowser-UNLOCKED",
                                 null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .
//Create a queue browser in unlocked mode
try {
  browser = admin.createClientQueueBrowser(client.getId());
} catch(BrokerException ex) {
  System.out.println("Failed to create client queue browser.");
  return;
}
. . .
//Browse operation
try {
  events = browser.browseEvents(max_events, msecs);
} catch(BrokerException ex) {
  System.out.println("Failed to complete browse operation.");
  return;
}
. . .
//Various browse operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close queue browser.");
  return;
}
```

Administratively Viewing Contents of a Locked Client Queue

For an administrative client to create a queue browser and examine client queue content in a locked mode, follow these steps.

1. Create a queue browser in locked mode using the BrokerAdminClient.createLockedClientQueueBrowser(String client_id) method. This will acquire a client queue lock to prevent further changes to the client queue. Note that even though the target

client queue is locked, the Broker will continue to queue incoming documents according to the client's subscription. "Acquiring a Queue Lock" on page 121 for instructions.

2. Set a filter or filters, if needed. It is important to note that once a filter is set for the queue browser, all browse operations will consider only documents that pass the filters. See "Filters on a Queue Browser" on page 122 for instructions.

3. Use browseEvents to view the content of the queue. See "BrokerQueueBrowser and BrokerClientQueueBrowser" on page 139 for instructions.

The following example illustrates how to examine a client queue in LOCKED mode:

```
BrokerClient client = null;
BrokerAdminClient admin = null;
BrokerLockedQueueBrowser browser = null;
BrokerEvent[] events = null;
int max_events = 100; //Maximum number of documents to be browsed
int msecs = 30000;   //Timeout for the browse operation

//Create a Broker client
try {
  client = new BrokerClient(broker_host,
             broker_name,
                           client_id,
                           client_group,
                           "QueueBrowser",
                           null);
} catch(BrokerException ex) {
  System.out.println("Failed to create client.");
  return;
}
//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                 broker_name,
                                null,
                                "admin",
                                "QueueBrowser-LOCKED",
                                null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .
//Create a locked queue browser
try {
  browser = admin.createLockedClientQueueBrowser(client.getId());
} catch(BrokerException ex) {
  System.out.println("Failed to create locked client queue browser.");
  return;
}
. . .

//Browse operation
try {
  events = browser.browseEvents(max_events, msecs);
} catch(BrokerException ex) {
  System.out.println("Failed to complete browse operation.");
  return;
```

```
}
. . .

//Various browse operations
. . .

//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close queue browser.");
  return;
}
```

Administratively Viewing Contents of an Unlocked Forwarding Queue

For an administrative client to examine forwarding queue content in an unlocked mode, follow these steps.

1. Create a queue browser in an unlocked mode using the BrokerAdminClient.createForwardQueueBrowser(Stringremote_broker_name) method. This will create a queue browser on the forwarding queue for the specified remote Broker.

   Stringremote_broker_name: Name of the remote Broker.

   Note:
   This mode of operation does not require a forwarding queue to be locked; therefore the forwarding queue content can undergo changes while the queue browser is open.

2. Set a filter or filters, if needed. Note that once a filter is set for the queue browser, all browse operations will only operate on documents that pass through the filters. See "Filters on a Queue Browser" on page 122 for instructions.

3. Use browseEvents to view the content of the queue. See "BrokerQueueBrowser and BrokerClientQueueBrowser" on page 139 for instructions.

The following example illustrates how to examine a forward queue in unlocked mode:

```
BrokerAdminClient admin = null;
BrokerQueueBrowser browser = null;
BrokerEvent[] events = null;
int max_events = 100; //Maximum number of documents to be browsed
int msecs = 30000;   //Timeout for the browse operation

//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                              null,
                              "admin",
                              "QueueBrowser-UNLOCKED",
                              null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
```

```
  return;
}
. . .
//Create a queue browser in unlocked mode
try {
  browser = admin.createForwardQueueBrowser(remote_broker_name);
} catch(BrokerException ex) {
  System.out.println("Failed to create forward queue browser.");
  return;
}
. . .
//Browse operation
try {
  events = browser.browseEvents(max_events, msecs);
} catch(BrokerException ex) {
  System.out.println("Failed to complete browse operation.");
  return;
}
. . .
//Various browse operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close queue browser.");
  return;
}
```

Administratively Viewing Contents of a Locked Forwarding Queue

For an administrative client to create a queue browser and examine forward queue content in a locked mode, follow these steps.

1. Create a queue browser in locked mode using the BrokerAdminClient.createLockedForwardQueueBrowser(Stringremote_broker_name) method. This will acquire a queue lock to prevent further changes to the forward queue. Note that even though the target queue is locked, the Broker will continue to queue incoming documents according to the remote Broker's subscription. “Acquiring a Queue Lock” on page 121 for instructions.

2. Set a filter or filters, if needed. Note that once a filter is set for the queue browser, all browse operations will only operate on documents that pass through the filters. See “Filters on a Queue Browser” on page 122 for instructions.

3. Use browseEvents to view the content of the queue. See “BrokerQueueBrowser and BrokerClientQueueBrowser” on page 139 for instructions.

The following example illustrates how to examine a forwarding queue in LOCKED mode:

```
BrokerAdminClient admin = null;
BrokerLockedQueueBrowser browser = null;
BrokerEvent[] events = null;
int max_events = 100; //Maximum number of documents to be browsed
int msecs = 30000;   //Timeout for the browse operation
```

```
//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                              null,
                              "admin",
                              "QueueBrowser-LOCKED",
                              null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .
//Create a locked queue browser on remote Broker
try {
  browser = admin.createLockedForwardQueueBrowser(remote_broker_name);
} catch(BrokerException ex) {
  System.out.println("Failed to create locked forward queue browser.");
  return;
}
. . .
//Browse operation
try {
  events = browser.browseEvents(max_events, msecs);
} catch(BrokerException ex) {
  System.out.println("Failed to complete browse operation.");
  return;
}
. . .
//Various browse operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close locked queue browser.");
  return;
}
```

Identifying "Unacknowledged" Documents

A queue browse operation may return documents including those that are retrieved but not yet acknowledged. These documents are "unacknowledged" or "unacked" documents. The document acknowledgment status information can be retrieved from the returned BrokerEvent object using the following methods.

- Use the BrokerEvent.getRetrievedStatus() method to check whether the document is an unacknowledged or not. If the document is unacked, this method returns "true".
- If the document is unacked, use the BrokerEvent.getRetrievedSessionId() method to get information on the client session that retrieved this document.

For more information about BrokerEvent, see the *webMethods Broker Client Java API Programmer's Guide*.

The following example illustrates how to examine an "unack" status on documents returned from a browse operation:

```
BrokerAdminClient admin = null;
BrokerLockedQueueBrowser browser = null;

//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                                null,
                                "admin",
                                "QueueBrowser",
                                null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .
//Create a queue browser
....

. . .
//Browse operation
try {
  events = browser.browseEvents(max_events, msecs);
} catch(BrokerException ex) {
  System.out.println("Failed to complete browse operation.");
  return;
}
//Check unacked status
try {
  for (int i=0; i< events.length; i++) {
    if (events[i].getRetrievedStatus()) {
      System.out.println("Event["+i+"] - "+events[i].getTypeName() "+
        "has been retrieved by client session id "+
      events[i].getRetrievedSessionId());
    }
    else {
      System.out.println("Event["+i+"] - "+events[i].getTypeName() "+
      " is available");
     }
  }
} catch(BrokerException ex) {
  System.out.println("Failed to complete ack status check on browse result.");
  return;
}
. . .
//Various browse operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close queue browser.");
  return;
}
```

Rearranging Queue Content

Rearranging queue content is possible only from a BrokerLockedQueueBrowser object.

Modifying Documents

Use the modifyEvents method to modify one or more documents on the target queue. A modify operation is equivalent to an "in-place replacement" of documents. You can only perform a modify operation from a BrokerLockedQueueBrowser object. Documents with "unack" status cannot be modified. The replacement document should match the storage type of the document that is being replaced.

The following example illustrates how to modify queue documents:

```
BrokerAdminClient admin = null;
BrokerLockedQueueBrowser browser = null;
BrokerEvent[] events = null;

//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                              null,
                              "admin",
                              "QueueBrowser-LOCKED",
                              null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .
//Create a locked queue browser on Broker client queue or forwarding queue
for a remote Broker
...
...

//Set cursor position to head
try {
  browser.setPosition(0);
 } catch(BrokerException ex) {
  System.out.println("Failed to complete setPosition operation.");
  return;
}
. . .
//Browse operation
try {
  events = browser.browseEvents(2, 10000);
} catch(BrokerException ex) {
  System.out.println("Failed to complete browse operation.");
  return;
}
//Swap events in position 0 and 1
try {
  long seqn[] = new long[2];
  BrokerEvent[] new_events = new BrokerEvent[2];

  seqn[0] = events[0].getReceiptSequenceNumber();
```

```
  seqn[1] = events[1].getReceiptSequenceNumber();
  new_events[0] = events[1];
  new_events[1] = events[0];

  browser.modifyEvents(seqn, new_events);
} catch(BrokerException ex) {
  System.out.println("Failed to complete modify operation.");
  return;
}
. . .
//Various browse operations
. . .
//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close queue browser.");
  return;
}
```

Inserting Documents

Use the BrokerLockedQueueBrowser. insertEventsAtHead method or insertEventsAtTail to insert one or more documents into the target queue.

The following example illustrates how to invoke the insertEvents method:

```
BrokerAdminClient admin = null;
BrokerLockedQueueBrowser browser = null;
BrokerEvent[] events = null;

//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                              null,
                              "admin",
                              "QueueBrowser-LOCKED",
                              null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
. . .
//Create a locked queue browser
....

//Browse operation
try {
  events = new BrokerEvent[2];
 events[0] = new BrokerEvent(client, "API::Test")
 events[1] = new BrokerEvent(client, "API::Test")
  //Populate event fields
  . . .

} catch(BrokerException ex) {
  System.out.println("Failed to create new events.");
  return;
```

```
}

//Insert at head operation
try {
  browser.insertEventsAtHead(events);
} catch(BrokerException ex) {
  System.out.println("Failed to complete insert at head operation.");
  return;
}
//Insert at tail operation
try {
  browser.insertEventsAtTail( events);
} catch(BrokerException ex) {
  System.out.println("Failed to complete insert at tail operation.");
  return;
}
. . .
//Various browse operations
. . .

//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close queue browser.");
  return;
}
```

Deleting Documents

Use the BrokerLockedQueueBrowser. deleteEvents method to remove one or more documents from the queue. In order to use this method, you must know the receipt sequence numbers of the documents that you are trying to delete. The receipt sequence number of a document is unique and is assigned when the document is placed in the queue.

You can use the BrokerEvent.getReceiptSequenceNumber method to access the receipt sequence number of a document. Documents with an "unacked" status can also be deleted using this API. For more information, see “Identifying "Unacknowledged" Documents” on page 132.

The following example illustrates how to delete client queue documents:

```
BrokerAdminClient admin = null;
BrokerLockedQueueBrowser = null;
BrokerEvent[] events = null;

//Create a Broker admin client
try {
  admin = new BrokerAdminClient(broker_host,
                broker_name,
                              null,
                              "admin",
                              "QueueBrowser-LOCKED",
                              null);
} catch(BrokerException ex) {
  System.out.println("Failed to create admin client.");
  return;
}
```

```
. . .
//Create a locked queue browser
...

//Browse operation
try {
  events = browser.browseEvents(max_events, msecs);
} catch(BrokerException ex) {
  System.out.println("Failed to complete browse operation.");
  return;
}
//Delete operation
try {
  long seqn[] = new long[events.length];

  for (int i = 0; i < events.length; ++i) {
    seqn[i] = events[i].getReceiptSequenceNumber();
  }
  browser.deleteEvents( seqn);
} catch(BrokerException ex) {
  System.out.println("Failed to complete delete operation.");
  return;
}
. . .
//Various browse operations
. . .

//Close the queue browser
try {
  browser.closeQueueBrowser();
} catch(BrokerException ex) {
  System.out.println("Failed to close client queue browser.");
  return;
}
```

Unblocking the Forwarding Queue in a Territory or Cluster

If you observe that an internal event in the forwarding queue for a remote Broker is blocking the queue and not allowing events to flow across, you must delete and recreate the remote Broker. If the Broker belongs to a territory or cluster, force the Broker to leave and then rejoin the territory or cluster.

When an internal event is blocking the forwarding queue, do the following to unblock the queue.

To unblock the forwarding queue in a territory or cluster

Note:
While changing a territory configuration or cluster configuration, allow some time for metadata changes to completely propagate across the network.

1. Create a backup of all the regular events contained in all the forwarding queues by saving the events to an external file. For more information about viewing events, see "Viewing Queue Content" on page 126, and *Administering webMethods Broker*.

2. Make the Broker leave the territory, or cluster. For more information, see "Removing a Broker from a Territory" on page 65.

3. Rejoin the Broker to the territory, or cluster. For more information, see "Adding a Broker to an Existing Territory" on page 65.

4. Restore all the regular events to their respective forwarding queues. For more information, see "Inserting Documents" on page 135.

Unblocking the Forwarding Queue in a Territory Gateway or Cluster Gateway

If you observe that an internal event in the forwarding queue for a remote Broker is blocking the queue and not allowing events to flow across, you must delete and recreate the remote Broker. If the Broker belongs to a territory gateway or cluster gateway, delete and recreate the blocked-half of the gateway.

To unblock the forwarding queue in a territory gateway or cluster gateway

Note:
While changing a territory gateway configuration or cluster gateway configuration, allow some time for metadata changes to completely propagate across the network.

1. Create a backup of all the regular events contained in all the forwarding queues by saving the events to an external file. For more information about viewing events, see "Viewing Queue Content" on page 126 and *Administering webMethods Broker*.

2. Save the gateway configuration. For more information about importing and exporting Broker metadata, see *Administering webMethods Broker*.

3. Delete the blocked-half of the gateway. For more information, see "Creating and Destroying Gateways" on page 68, and "Creating and Destroying Cluster Gateways" on page 79.

4. Reconfigure the gateway. For more information about importing and exporting Broker metadata, see *Administering webMethods Broker*.

5. Restore all the regular events to their respective forwarding queues. For more information, see "Inserting Documents" on page 135.

   Note:
   Because a Broker in a territory or cluster can have multiple forwarding queues, the backup of the forwarding queue events must be restored (reinserted) in the correct forwarding queue.

BrokerAdminClient

The BrokerAdminClient class is used to represent a Broker administrative client. For more information about the BrokerAdminClient class, see "Managing Broker Clients" on page 25.

You use the createClientQueueBrowser and createLockedClientQueueBrowser methods in the BrokerAdminClient class to create queue browsers on a specific client in unlocked and locked modes, respectively.

You use the createForwardQueueBrowser method to create a queue browser on a specific forward queue for a remote Broker in unlocked mode and createLockedForwardQueueBrowser method in locked mode.

BrokerClient

The BrokerClient class is used to represent a Broker client. For more information about the BrokerClient class, see "Managing Broker Clients" on page 25.

You use the createClientQueueBrowser method in BrokerClient to create a queue browser on the client's own queue. Once you create a queue browser from a client session, that session will be marked as a queue browsing session and cannot be used to retrieve documents from the queue using regular document retrieval methods. Hence, wherever possible, you must create a queue browser on a separate client session.

Note:
You cannot use the BrokerClient class to create a queue browser on the forwarding queue for a remote Broker; use the BrokerAdminClient class instead.

BrokerQueueBrowser and BrokerClientQueueBrowser

Note: BrokerQueueBrowser is an interface for queue browser methods, implementations of which allow you to browse both client queue and forwarding queue. BrokerQueueBrowser interface has two implementations, BrokerClientQueueBrowser for browsing client queues and BrokerForwardQueueBrowser for forwarding queues.

You use the BrokerQueueBrowser interface to access all the methods that you can use to browse a queue.

An application can use the BrokerQueueBrowser interface to perform the following functions on the client queue:

- Browse documents in the client's queue.
- Selectively browse documents using filters.

An application can use the BrokerQueueBrowser interface to perform the following functions on the forwarding queue for the remote Broker:

- Browse documents in the forwarding queue for the remote Broker queue.
- Selectively browse documents using filters.

All methods throw BrokerQueueBrowserException (and other exceptions) when necessary.

The BrokerQueueBrowser interface methods are listed in the following table.

Method	Description
getBrowserInfo	Get browser information of the current queue browser.
setPosition	Set the browse cursor to the specified position in the queue. **Note:** While setting the position of the browse cursor in a forwarding queue, keep in mind that the queue might also contain Broker internal events. For more information, see "Forwarding Queue Browser" on page 114.
setFilter	Set up a single filter for queue browse operations. If the queue browser already has one or more filters, this new filter is added to the list. If a filter is already present for a document type, the existing filter will be overwritten with the new filter. When one or more filters are set on the queue browser, a browse operation only returns documents matching those filters. Note that queue browser filters are applicable only to the browse operation and not to other queue browser operations, such as insert or delete operations.
setFilters	Set up a multiple filters for queue browse operations. If the queue browser already has one or more filters, the new filters are added to the list. If a filter is already present for a document type, the existing filter will be overwritten with the new filter. When filters are set on the queue browser, a browse operation only returns documents matching those filters. Note that queue browser filters are applicable only to the browse operation and not to other queue browser operations, such as insert or delete operations.
resetFilters	Reset the filter list on the current queue browser. The filter list on the queue browser will be emptied out.
getFilters	Get filters that are currently set for queue browse operation. The existing filters are returned in the form of BrokerFilterDescriptor objects.
browseEvents	Read one or more documents from the queue on which the queue browser is opened. This method simply returns documents from the current queue position. The number of documents being returned is not guaranteed to be the requested number of documents, even if the queue contains more than that many documents. Any number of documents up to the value of max_events may be returned. **Note:** While returning the documents, the queue browser skips all the internal events that might be queued in a forwarding queue. For more information, see "Forwarding Queue Browser" on page 114.

Method	Description
closeQueueBrowser	Close the queue browser and release the associated resources.

BrokerLockedQueueBrowser and BrokerLockedClientQueueBrowser

Note: BrokerLockedQueueBrowser is the interface for queue browser methods, implementations of which allow you to browse both client queue and forwarding queue. BrokerLockedQueueBrowser interface has two implementations, BrokerLockedClientQueueBrowser for browsing client queues and BrokerLockedForwardQueueBrowser for forwarding queues.

BrokerLockedQueueBrowser extends the BrokerQueueBrowser interface. In addition to browsing the queue, the BrokerLockedQueueBrowser interface allows you to manipulate the contents of the queue.

An application might use the BrokerLockedQueueBrowser interface to perform the following functions on the forwarding queue for a remote Broker, and the client queue.

- Obtain a queue lock to manage a client queue.
- Browse documents in the locked client's queue.
- Modify documents in the locked client's queue.
- Insert documents into the locked client's queue.
- Delete documents from the locked client's queue.
- Set filters for browse and delete operations.
- Release an acquired queue lock.

All methods throw BrokerQueueBrowserException (and other exceptions) when necessary.

The additional methods that are available only on BrokerLockedQueueBrowser are listed in the following table.

Method	Description
`modifyEvents`	Modify documents in the queue. The documents are identified by the specified receipt sequence numbers for an in-place replacement in the queue.
`deleteEvents`	Delete documents in the queue. The documents are identified by the receipt sequence numbers.
`insertEventsAtHead`	Insert documents into the queue at head position.
`insertEventsAtTail`	Insert documents into the queue at tail position.

All methods throw BrokerQueueBrowserException (and other exceptions) when necessary.

A API Exceptions

This appendix describes the exceptions that are unique to the webMethods Broker Java administrative API. Other possible exceptions are described in the *webMethods Broker Client Java API Programmer's Guide*.

BrokerDependencyException
Your request to destroy a client group or an event type (without the *force--destroy* option) could not be completed because there are Broker clients that depend on the group or event type.

BrokerExistsException
You have attempted to create a Broker with a name that is already being used by another Broker on the same Broker Server.

BrokerInClusterException
You have attempted to destroy a Broker that is currently part of a cluster or you have attempted to add a Broker to a cluster when it is already a member of a cluster.

BrokerIncompatibleVersionException
You have attempted to perform an operation on a Broker Server that is not supported. This results from the Broker Server in question having an earlier version number than the API that your client is using.

BrokerInTerritoryException
You have attempted to destroy a Broker that is currently part of a territory or you have attempted to add a Broker to a territory when it is already a member of a territory.

BrokerInvalidAccessListException
The Access Control List you specified is not valid.

BrokerInvalidBrokerNameException
The Broker name you specified contains illegal characters. See the appendix in the *webMethods Broker Client Java API Programmer's Guide* for details on naming rules.

BrokerInvalidClientGroupNameException
The client group name you specified contains illegal characters. See the appendix in the *webMethods Broker Client Java API Programmer's Guide* for details on naming rules.

BrokerInvalidLogConfigException
The log configuration object you have specified is not valid.

BrokerInvalidClusterNameException

The cluster name you specified when creating a cluster is not valid. See the appendix in the *webMethods Broker Client Java API Programmer's Guide* for details on naming rules.

BrokerInvalidNameException
The application name you specified when creating a Broker client is not valid. See the appendix in the *webMethods Broker Client Java API Programmer's Guide* for details on naming rules.

BrokerInvalidTerritoryNameException
The application name you specified when creating a territory is not valid. See the appendix in the *webMethods Broker Client Java API Programmer's Guide* for details on naming rules.

BrokerNotInClusterException
You have attempted to create or destroy a cluster gateway, but the Broker you have specified is not a member of any cluster.

You have specified a Broker that is not a member of any territory when attempting to add a Broker to or remove a Broker from a territory.

This error may also occur if you attempt to list the territories, territory gateways, shared event types, or territory statistics for a Broker that is not a member of any territory.

BrokerNotInTerritoryException
You have attempted to create or destroy a territory gateway, but the Broker you have specified is not a member of any territory.

You have specified a Broker that is not a member of any territory when attempting to add a Broker to or remove a Broker from a territory.

This error may also occur if you attempt to list the territories, territory gateways, shared event types, or territory statistics for a Broker that is not a member of any territory.

BrokerUnknownBrokerNameException
You have attempted to create or modify a territory gateway, but the remote half of the gateway has not yet been established and the remote Broker could not be found. You should start the remote Broker and attempt the operation again.

BrokerUnknownClusterException
You have attempted to create or modify a cluster, but the cluster with the name you specified cannot be found.

This error might also occur if Broker did not find a gateway to the cluster you specified.

BrokerUnknownTerritoryException
You have attempted to create or modify a territory, but the specified territory cannot be found.

This error might also occur if Broker did not find a gateway to the territory you specified.