

webMethods Broker Client C API Programmer's Guide

Version 10.15

October 2022

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-C-CLIENT-PROGRAMMERS-GUIDE-1015-20220912

Table of Contents

About this Guide	7
Document Conventions.....	8
Online Information and Support.....	9
Data Protection.....	10
1 Getting Started	11
Overview.....	12
Overview of Event-Based Applications.....	12
Using webMethods Broker C API.....	15
2 Creating and Initializing Events	19
Event Overview.....	20
Creating Events.....	22
Deleting Events.....	23
Converting Events as Binary Data.....	23
Event Data Fields.....	23
Regular Data Fields.....	25
Sequence Data Fields.....	27
Structure Data Fields.....	30
Envelope Fields.....	32
Specifying Field Names.....	37
3 Using Broker Clients	41
Understanding Broker Clients.....	42
Creating and Destroying Broker Clients.....	43
Disconnecting and Reconnecting.....	46
Connection Notification.....	49
Obtaining Client State and Status.....	50
Advanced Features.....	51
Broker Connection Descriptors.....	53
4 Subscribing to and Receiving Events	59
Overview.....	60
Event Subscriptions.....	60
Receiving Events in the Get-Events Model.....	62
5 Publishing and Delivering Events	69
Overview.....	70
Publishing Events.....	70
Delivering Events.....	72
6 Using the Callback Model	75

Overview.....	76
Understanding Callbacks.....	76
General Callback Functions.....	78
Specific Callback Functions.....	79
Dispatching Callback Functions.....	80
7 Transaction Semantics.....	83
Overview.....	84
Transactional Client Processing.....	84
Using Transaction Processing.....	85
8 Using Request-Reply.....	91
Overview.....	92
The Request-Reply Model.....	92
The Requestor.....	94
The Server.....	98
9 Handling Errors.....	105
Overview.....	106
Using BrokerErrors.....	106
Setting the System Diagnostic Level.....	108
10 Using BrokerDate Objects.....	109
Overview.....	110
Date and Time Representation.....	110
BrokerDate Functions.....	110
11 Managing Event Types.....	113
Overview.....	114
Event Type Definitions.....	114
Event Type Definition Cache.....	117
Infosets.....	118
12 Configuring Broker Client Security.....	121
Overview.....	122
Working with Basic Authentication.....	122
Working with Secure Socket Layer (SSL).....	124
13 Using Event Filters.....	129
Overview.....	130
Filter Strings.....	130
Using Filters with Subscriptions.....	141
Using BrokerFilters.....	142
14 API Reference.....	145
awAbort.....	148

awAcknowledge.....	148
awBegin.....	150
awBroker.....	151
awCan.....	153
awCancel.....	154
awClear.....	161
awClient.....	163
awCommit.....	164
awCompare.....	164
awCopy.....	165
awDate.....	166
awDelete.....	167
awDeliver.....	169
awDescriptor.....	182
awDestroy.....	183
awDisconnect.....	184
awDispatch.....	186
awDoes.....	187
awEnd.....	188
awError.....	189
awEvent.....	190
awFilter.....	197
awFree.....	197
awFlush.....	198
awGet.....	198
awInit.....	279
awInterrupt.....	279
awIs.....	280
awLock.....	284
awMain.....	284
awMake.....	285
awMalloc.....	289
awMatch.....	290
awNative.....	290
awNew.....	291
awParse.....	311
awPrime.....	313
awPublish.....	314
awRealloc.....	319
awReconnect.....	319
awRegister.....	322
awResend.....	327
awSet.....	327
awSSL.....	362
awStop.....	362
awString.....	363
awThreaded.....	371
awTx.....	372
awType.....	394
awUnlock.....	394

awValidate.....	394
15 Using Sequence Numbers.....	397
Overview.....	398
Sequence Numbers.....	398
Publisher Sequence Numbers.....	398
Receipt Sequence Numbers.....	399
A Error Definitions.....	403
B Parameter Naming Rules.....	407
Overview.....	408
Length Restriction.....	408
Restricted Characters.....	408
Reserved Words.....	409
System Parameters.....	409
Broker Parameter Restrictions.....	409
C Unicode String Functions.....	411
Overview.....	412
Basic Unicode String Functions.....	412
Unicode String Conversion Functions.....	415
Unicode String Copy/Conversion Functions.....	417
Unicode String Length Functions.....	420
D Transactional Client Processing with Adapters.....	423
Overview.....	424
Transaction Processing.....	424
Using Transaction Processing.....	428

About this Guide

■ Document Conventions	8
■ Online Information and Support	9
■ Data Protection	10

The *webMethods Broker Client C API Programmer's Guide* describes the application programming interfaces (API) that you use to create event-based applications with the C programming language.

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 C programming language.

This book assumes you are familiar with the terminology and basic operations of your operating system (OS). If you are not, please refer to the appropriate documentation for that OS.

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 <i>folder.subfolder.service</i> , APIs, Java classes, methods, properties.
<i>Italic</i>	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	12
■ Overview of Event-Based Applications	12
■ Using webMethods Broker C API	15

Overview

This chapter describes the basic features of the C language API for the webMethods Broker system. After reading this chapter, you should be ready to begin learning how to develop Broker applications.

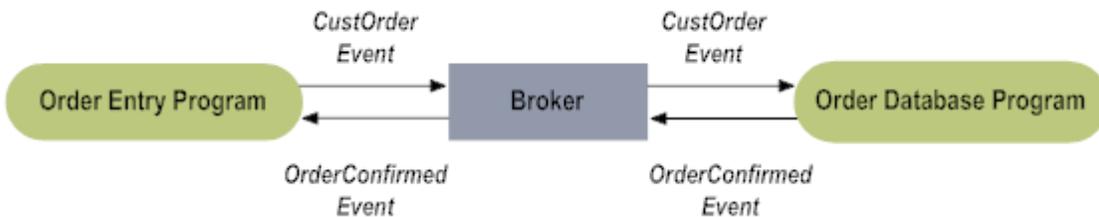
Overview of Event-Based Applications

The webMethods Broker system provides you with tools and libraries for building powerful, event-based applications that are made up of de-coupled *client programs*. The client programs are de-coupled because they communicate by sending and receiving *events* through a third entity called a *Broker*. A Broker is actually part of a *Broker Server*, which may contain multiple Brokers.

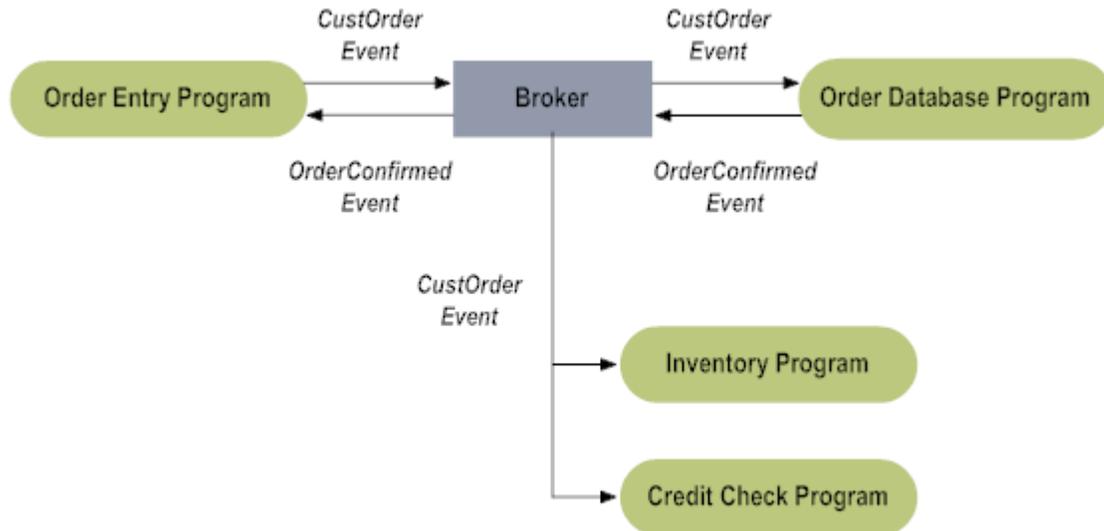
The figure below shows a simple example where an Order Entry program receives input from a customer and *publishes* a CustOrder event. The act of publishing the event causes the event to be sent to the Broker. The Broker then determines that the Order Database program has *subscribed* to the CustOrder event, and so the Broker distributes the event to the Order Database client program.



More complex applications are possible than the simple example shown in figure below. The Order Database program could create an OrderConfirmed event that would be sent to the Order Entry program as a reply to a CustOrder event.



You might also design a system where several client programs receive the CustOrder event and perform their own processing, such as checking the inventory on hand or verifying the customer's credit status.



The webMethods Broker system also allows the Broker and the client programs that comprise an application to execute on different hosts within a network.

Events

Events are one of the most important ingredients in an application developed with webMethods Broker. Events are defined using Software AG Designer, described in the *Software AG Designer Online Help*.

Note: Broker events are known as *Broker document types* in Designer.

Events have these important characteristics:

- Event types are organized into *event scopes*, allowing you to group the events related to your application.
- Events have a unique *event type*, which define the *event fields* they contain and the data type associated with each field.
- Event fields represent data in a platform-independent representation, allowing clients on different platforms to exchange information.

Events are discussed in greater detail in [“Creating and Initializing Events” on page 19](#).

Broker Clients

Client programs may consist of one or more *Broker clients*. Just as a program can open and use more than one file, a program may create and use multiple Broker clients. Once created, a Broker client represents a connection to a particular Broker on a particular host. A Broker client can subscribe to events, publish events, and receive events. Broker clients may also share a connection to a Broker and they may also share the same client queue and client state. Broker clients are covered in greater detail in [“Using Broker Clients” on page 41](#).

The Broker

The Broker coordinates the exchange of events between client programs. To accomplish this complex task, the Broker:

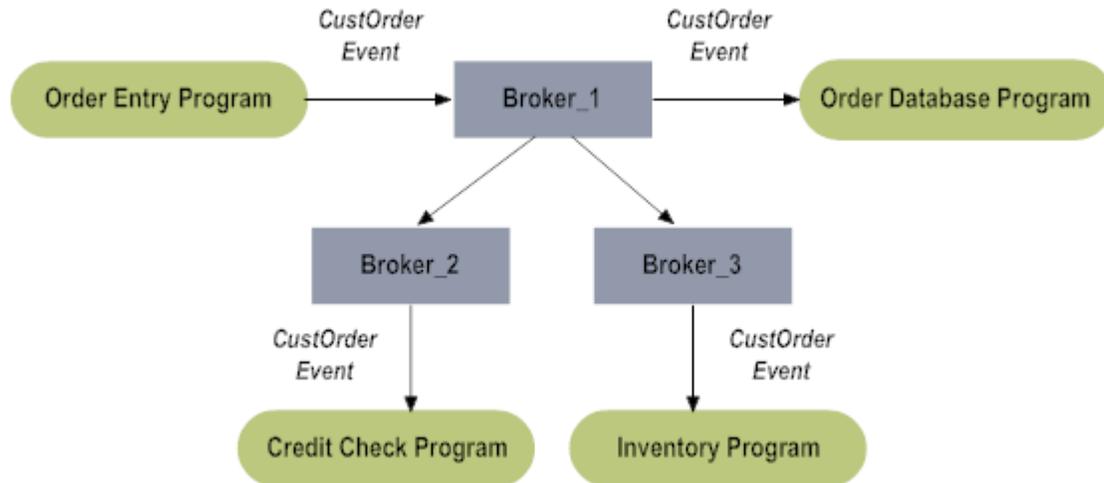
- Queues all events that are published by Broker clients.
- Sends events to those clients which have subscribed to and are ready to receive the event.
- Provides various levels of reliable delivery of events through the use of event sequence numbers. Sequence numbers are described in [“Managing Event Types” on page 113](#).
- Provides event filtering services that allow Broker clients to selectively filter the events they will receive, based on event content. See [“Using Event Filters” on page 129](#) for more information on filters.
- Maintains information on all event type definitions, through Designer. See the *Software AG Designer Online Help* for more information.

Note: Broker events are known as *Broker document types* in Designer.

- Maintains information on *client groups*, such as event publication permissions, event subscription permissions, network access control, and client event queue characteristics for Broker clients in each group.
- Maintains an event queue and client state information for each Broker client that has been created.

Broker-to-Broker Communication

The webMethods Broker system 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 next figure 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.



In order to share information and forward events, Brokers must join a *territory*. All Brokers within the same territory have knowledge of one another's event type definitions and client groups. For more information, see *Administering webMethods Broker*.

Using webMethods Broker C API

The webMethods Broker C API is implemented as a set of C library files and header files. The library provides all the necessary functions for:

- Creating and manipulating events.
- Transferring events to the Broker.
- Retrieving events from the Broker.
- Querying the Broker for state information.

You can use the webMethods Broker C API to create a wide variety of applications, including simple clients, watchdog agents, and adapters for legacy applications and existing data sources.

Most UNIX platforms have the following library-file object modes:

- **LP64 object mode.** In this mode, the "pointers" and the data type "long" are 64-bit values. If you compile your applications in the LP64 mode, you must use the LP64 API. The LP64 version of the C API libraries are available in *webMethods Broker_directory/lib* directory.
- **LP32 object mode.** In this mode, the "pointers" and the data type "long" are 32-bit values. If you compile your applications in the LP32 mode, you must use the LP32 API. If you have both C API versions (that is, LP64, and LP32) installed, the LP32 version of the C API libraries are available in the *webMethods Broker_directory/lib32* directory.

Compatibility

This version of webMethods Broker is compatible with applications and Brokers from webMethods Broker 9.x and later.

Though webMethods Broker is written in C, applications compiled with a version of C++ that support calls to C language routines may also use the API.

Applications developed with webMethods Broker are fully interoperable with applications developed with the Broker Java API version 9.x or later.

Application Development with webMethods Broker

Using webMethods Broker system to develop applications involves the following steps:

1. Install the webMethods Broker system. See *Installing and Upgrading webMethods Broker* or your system administrator.
2. Start the Broker. See *Installing and Upgrading webMethods Broker* or your system administrator.
3. Design the client programs and the events that will comprise your system.
4. Use Designer to define the events (or documents) for your application. See the *Software AG Designer Online Help* or your system administrator.

Note: Broker events are known as *Broker document types* in Designer.

5. Write and compile your client code, linking it with the webMethods Broker C library.
6. Execute your client programs.

webMethods Broker Client Include Files

These include files may be used in either C or C++ programs. For simplicity, you may include the `aweb.h` file in your code, which will include all of the other include files.

File name	Description
<code>awclient.h</code>	Defines client functions.
<code>awcommon.h</code>	Contains general platform definitions.
<code>awdate.h</code>	Defines BrokerDate functions.
<code>awdesc.h</code>	Defines BrokerConnectionDescriptor functions.
<code>aweb.h</code>	Includes all of the following include files for you.
<code>awerror.h</code>	Defines error accessor functions.
<code>awerror_msg.h</code>	Defines error codes.
<code>awetdef.h</code>	Defines event type definition functions.
<code>awevent.h</code>	Defines event access and manipulation functions.
<code>awfilter.h</code>	Defines event filtering functions.

File name	Description
awintl.h	Defines string manipulation routines to support internationalization.
awlong.h	Defines BrokerLong functions.
awqlock.h	Defines client queue locking functions.
awstring.h	Defines BrokerString functions.
awtypes.h	Defines basic types and values.
awtxclient.h	Defines transactional Broker client functions.

- On UNIX platforms, these include files that can be found in *webMethods Broker_directory* /include.
- On Windows platforms, these include files can be that found in *webMethods Broker_directory* \include.

Note:

Use `awFree()` to release memory allocated by the webMethods Broker C API.

webMethods Broker C API Library Files

The following libraries are provided for webMethods Broker C API:

File name in UNIX	File name in Windows	Description
libawc_static.a	awc_static.lib	Thread-safe, static library containing C API functions.
libawssl_static.a	awssl_static.lib	Thread-safe, static library containing internal SSL support functions.
libcrypto.a	libeay32.lib	Thread-safe, OpenSSL library.
libssl.a	ssleay32.lib	Thread-safe, OpenSSL library.

The library files are available in *webMethods Broker_directory* \lib and the dll files are available in *webMethods Broker_directory* \bin. The LP32 version of the C API libraries are available in the *webMethods Broker_directory* /lib32 directory.

2 Creating and Initializing Events

■ Event Overview	20
■ Creating Events	22
■ Deleting Events	23
■ Converting Events as Binary Data	23
■ Event Data Fields	23
■ Regular Data Fields	25
■ Sequence Data Fields	27
■ Structure Data Fields	30
■ Envelope Fields	32
■ Specifying Field Names	37

Event Overview

The first step in implementing your application is to define the content of the events that will be used to communicate information between your client applications and the Broker. Events are defined using Software AG Designer, described in the *Software AG Designer Online Help*.

Note: Broker events are known as *Broker document types* in Designer.

Events contain data that allow applications to communicate with each other. An event may be used to represent:

- Requests to a database application to retrieve or write data.
- Banking transactions such as deposits, withdrawals, and transfers.
- News headlines, stock quotes, or money exchange rates.
- Customer orders, invoices, packing slips, or credit memos.

Event Types

An *event type* defines the event's name as well as the fields that it contains. Both the Broker and the webMethods Broker library functions use event type definitions to verify that an event's fields are set with values that match their defined types. Event type definitions are also used if you call the [awValidateEvent](#) function to validate an event's fields.

Note:

Events created without a `Broker client` context are not type checked. See [“Field Type Checking” on page 22](#) for more information.

Event Type Cache

The webMethods Broker library copies the event type definitions used by your application from the Broker to which you are connected into a local *event type cache*. The cache improves the performance of the event field type checking process and its use is usually transparent to you. For more detailed information, see [“Managing Event Types” on page 113](#).

Event Type Names

Each event type has a unique name that distinguishes it from other event types.

Note: [“Parameter Naming Rules” on page 407](#) describes the restrictions on event type names.

Event types are organized into event families, allowing you to group all of the events related to a particular application domain. An event type name consists of two components; a *scope* and a *base name*.

Scope: :BaseName

The scope component can consist of one or more levels. Consider the following fully qualified event type name.

```
WesternRegion::Hardware::Sales::receiveCustOrder
```

The base name for this event type would be `receiveCustOrder`.

The scope would be `WesternRegion::Hardware::Sales`.

Use the [awGetEventTypeName](#) method to obtain the fully qualified name for an event type

Use the [awGetEventTypeNameBaseName](#) to obtain the base name for an event type.

Use the [awGetEventTypeNameNames](#) to obtain the fully qualified names of all the event types known to the Broker to which your Broker client is connected.

Use the [awGetScopeNames](#) to obtain the fully qualified names of all the event types known to the Broker to which your Broker client is connected. All event scopes that contain at least one event type will be returned.

Note:

Only the names of the event types which your client is permitted to browse are returned by the [awGetEventTypeNameNames](#) and [awGetScopeNames](#) functions. In most cases, this corresponds to the set of event types which your client can publish or for which it can register subscriptions.

Event Fields

Every event contains *envelope* fields and *data* fields. Envelope fields are consistent for all event types and contain details about the event's sender, destination, and its transit. Envelope fields are described on [“Envelope Fields” on page 32](#).

Event data fields contain the data that your client applications use to exchange information. Event data fields may contain a single value, a sequence of values with the same type, or a structure containing values of different types. Event data fields are discussed on [“Event Data Fields” on page 23](#).

Note: [“Parameter Naming Rules” on page 407](#) describes the restrictions on event field names.

Event Identifier

When an event is published, described in [“Publishing and Delivering Events” on page 69](#), the Broker will assign the event an event identifier that you can use to determine if two events are exactly the same. You can also use the event identifier to match events with trace events or activity traces, described in *Administering webMethods Broker*. An event identifier is almost certainly unique. It is possible, though extremely unlikely, for two Brokers to generate the same event identifier.

You can retrieve the event identifier using the [awGetEventId](#) method.

Creating Events

Before your client application can publish or deliver an event, it must create the event and set the event's fields. The following example contains an excerpt from the `publish1.c` sample application that shows the use of the [awNewBrokerEvent](#) function to create an event. The function takes the following parameters:

- A `Broker client` pointer.
- The name of the event type. In this example, the event scope is "Sample" and the event type name is "SimpleEvent."
- A `BrokerEvent` that, upon return, will reference the newly created event.

The following example illustrates how to create an event:

```
int main(int argc, char **argv)
{
    Broker client c;
    BrokerEvent e;
    . . .
    /* Create the event */
    . . .
    err = awNewBrokerEvent(c, "Sample::SimpleEvent", &e);
    if (err != AW_NO_ERROR) {
        printf("Error on create event\n%s\n", awErrorToString(err));
        return 0;
    }
    . . .
}
```

General Event Functions

After creating an event, you may use the [awCopyEvent](#) function to create a new event, copying the contents of an existing event to the new event.

You may also use the [awErrorToString](#) function to convert the fields of an event to a string that is suitable for display.

The [awEventToFormattedString](#) function also lets you convert the fields of an event to a string that is suitable for display. Unlike the [awEventToString](#) function, this function lets you control the format of the resulting string. This function uses the C language locale to convert event fields that have a data type of `double`, `float`, or `BrokerDate`.

The [awEventToLocalizedString](#) uses the current locale to convert all of an event's fields into a string that is suitable for display.

Use the [awGetEventClient](#) to obtain the `Broker client` associated with an event.

Field Type Checking

When you create an event with a valid `Broker client` parameter, the following field type checking rules will be applied to the event.

1. All event fields will appear to be set on the event at the time the event is created.
2. You are not allowed to set a field which does not exist for the event type.
3. You cannot set an event field with a type other than that defined by the event type.

When you create an event with a `NULL BrokerClient` parameter, the following type checking rules will be applied to the event.

1. You may set fields with any field name and with any data type.
2. Any attempt to retrieve an event field that was not previously set will return an `AW_ERROR_FIELD_NOT_FOUND` error.
3. Once a field has been set, you are not allowed to change the field's data type without first clearing the field, using the `awClearEventField` function, or clearing the entire event, using the `awClearEvent` function.

Note:

For most applications, you should create an event within the context of a Broker client so that type checking will occur.

Deleting Events

The following example contains an excerpt from the `publish1.c` sample application that shows the use of the `awDeleteEvent` function to delete an event.

```

. . .
    awDeleteEvent(e);
    return 0;
}

```

Converting Events as Binary Data

Use the [awEventToBinData](#) function to obtain a binary array representation of a `BrokerEvent` that is suitable for saving to disk.

Restoring Events from Binary Data

Use the [awEventFromBinData](#) function to initialize a `BrokerEvent` from a binary array that was generated from an earlier call to [awEventToBinData](#). When re-creating the event, you may specify an associated Broker client if you wish the event's contents to be type checked.

Event Data Fields

Data fields contain application-specific data that your application will set before publishing an event or will retrieve when processing a received event. Each field has a name and a specifically typed value. Event data field values are stored in a platform-independent representation that allows client applications executing on different hardware platforms and under different operating systems to easily exchange data without worrying about bit significance or byte ordering. The

conversion from the webMethods Broker platform-independent data representation to the local data representation is handled transparently by the Broker get and set functions.

Field Data Types

The following table shows the data types used by client applications for regular and sequence data fields.

Event or Editor Type	C Language Type	Description
byte	char	Signed 8-bit integer.
short	short	Signed 16-bit integer.
int	long	Signed 32-bit integer.
long	BrokerLong	Signed 64-bit integer.
float	float	Standard-precision floating point number.
double	double	Double-precision floating point number.
boolean	BrokerBoolean	1 (true) or 0 (false), stored as a single byte.
date	BrokerDate	A structure representing the year, month, day, hour, minute, second, and millisecond.
string	BrokerString	A string of characters.
char	char	A single character.
unicode_char	charUC	A Unicode character.
unicode_string	charUC *	A Unicode string.

Use the [awGetEventFieldType](#) function to obtain an event field's type.

Note:

See “Unicode String Functions” on page 411 for a description of the functions offered for platforms that do not provide built-in support for Unicode characters and strings.

Obtaining Field Names

You can use the [awGetFieldNames](#) function to obtain the names of the fields contained in a particular event.

Determining if a Field is Set

You can use the [awIsEventFieldSet](#) function to determine if a field within a particular event has been set.

Regular Data Fields

A regular data field in an event contains a single value that is obtained and set using the field name and an appropriately typed source or target value.

Setting Regular Data Fields

Several functions are provided for setting a regular data field, based on its type. These functions are described in [awSet<type>Field](#). The source value for the field being set depends on the field's type, as shown below.

Function Name	Value Type
awGetUCStringFieldAsUTF8	BrokerBoolean
awSetByteField	char
awSetCharField	char
awSetDate	BrokerDate
awSetDoubleField	double
awSetFloatField	float
awSetIntegerField	long
awSetLongField	BrokerLong
awSetShortField	short
awSetStringField	char *
awSetUCCharField	charUC
awSetUCStringField	charUC *

The following example contains an excerpt from the `publish1.c` sample application that shows the use of the `awSetIntegerField` function. The function takes the following parameters:

- A `BrokerEvent` reference.
- The name of the event field to be set.
- The value for the field.

```
long count;
. . .
count = 1;
err = awSetIntegerField(e, "count", count);
if (err != AW_NO_ERROR) {
    printf("Error on setting event field\n%s\n",
        awErrorToString(err));
}
```

```
    return 0;
}
```

If you attempt to set a field with a value that does not match its defined type, an error may be returned. Type checking will not occur if the event whose field is being set was created without a Broker client context, as described on [“Field Type Checking” on page 22](#).

Getting Regular Field Values

Several functions are provided for obtaining the value of a regular event data field, based on its type. These functions are described in [awGet<type>Field](#). The type of the value being retrieved depends on the field's type, as shown below.

Function Name	Value Type
awGetBooleanField	BrokerBoolean
awGetByteField	char
awGetCharField	char
awGetDateField	BrokerDate
awGetDoubleField	double
awGetFloatField	float
awGetIntegerField	long
awGetLongField	BrokerLong
awGetShortField	short
awGetStringField	char *
awGetUCCharField	charUC
awGetUCStringField	charUC *

The following example contains an excerpt from the `subscribe1.c` sample application that shows the use of the `awGetIntegerField` function. The function takes the following parameters:

- A `BrokerEvent` reference.
- The name of the event field to get.
- The pointer to a location to store the value retrieved from the field.

```
long pub_count;
. . .
err = awGetIntegerField(e, "count", &pub_count);
if (err != AW_NO_ERROR) {
    printf("Error on getting count field\n%s\n", awErrorToString(err));
    return 0;
}
```

```

} else {
    printf("Event #%d received with count=%d\n",
        count, pub_count);
}
. . .

```

If you attempt to get a field value with a type that does not match the field's defined type, an error may be returned. Type checking will not occur if the event whose field is being obtained was created without a Broker client, as described on “[Field Type Checking](#)” on page 22.

You may also use the [awGetField](#) to obtain the value and type of an event field.

Unicode Transform Function 8

The webMethods Broker API allows you to retrieve and set Unicode string fields that use the *Unicode Transform Function 8* (UTF-8) encoding. UTF-8 allows 16-bit Unicode characters to be represented as a sequence of up to four 8-bit values. The following functions are provided:

- [awGetUCStringFieldAsUTF8](#)
- [awSetUCStringFieldAsUTF8](#)

Sequence Data Fields

Sequence data fields contain a sequence of several values with the same data type. A sequence field can contain any of the types mentioned for simple data fields. A multiple dimension array of values may be represented as a sequence within a sequence.

Setting Sequence Fields

Several functions are provided for setting the values of the entire sequence, or a subset of the sequence. These functions are described in [awSet<type>SeqField](#).

You may also use the [awSet<type>Field](#) to set a single value within a sequence field by specifying the field name with an index.

The following example shows the use of the [awSetIntegerSeqField](#) function. The function takes the following parameters:

- A BrokerEvent reference.
- The name of the event sequence field to be set.
- The number of elements to skip from the beginning of the source array parameter.
- The number of elements to skip from the beginning of the target sequence in the event.
- The number of elements to be set.
- The source array of values of the appropriate type.

```

long count[5] = { 0, 1, 2, 3, 4};
long count_ptr = &count;

```

```

. . .
err = awSetIntegerSeqField(e, "countSeq", 0, 0, 5, count_ptr);
if (err != AW_NO_ERROR) {
    printf("Error on setting event sequence field\n%s\n",
        awErrorToString(err));
    return 0;
}
. . .

```

The manner in which the source array is specified depends on the array's type.

Function Name	Value Type
awSetBooleanSeqField	BrokerBoolean *
awSetByteSeqField	char *
awSetCharSeqField	char *
awSetDateSeqField	BrokerDate *
awSetDoubleSeqField	double *
awSetFloatSeqField	float *
awSetIntegerSeqField	long *
awSetLongSeqField	BrokerLong *
awSetShortSeqField	short *
awSetStringSeqField	char **
awSetUCCharSeqField	charUC *
awSetUCStringSeqField	charUC **

Each of the `awSet<type>SeqField` functions may overwrite all or part of the destination sequence field. After the code shown in the example on [“Setting Sequence Fields” on page 27](#) is executed, the sequence will contain:

```
[0 1 2 3 4]
```

If you then set three elements (3, 2, 1) into this same sequence at location 1, the sequence would then appear as:

```
[0 3 2 1 4]
```

An error will be returned if you attempt to set a field with a value that does not match its defined type.

These functions may also cause the destination sequence to grow in size, if a larger number of elements are stored into the sequence.

These functions *never* reduce the number of elements in the destination sequence: Use the [awSetSequenceFieldSize](#) function to reduce the size of a sequence field.

Getting Sequence Field Values

Several functions are provided for obtaining all of the values from a sequence event field, or a subset of the sequence, with a single function call. These functions are described in [awGet<type>SeqField](#).

You may also use the [awGet<type>Field](#) to obtain a single value from a sequence field by specifying the field name with an index.

The following example shows the use of the [awGetIntegerSeqField](#) function. The function takes the following parameters:

- A BrokerEvent reference.
- The name of the event sequence field being accessed.
- The number of elements to skip from the beginning of the sequence field in the event.
- The number of source elements to be retrieved.
- An integer where the number of retrieved elements is stored.
- The address of a pointer where the values can be stored.

```
long *count_array;
long num_retrieved;
. . .
err = awGetIntegerSeqField(e, "countSeq", 0, 5,
    &num_retrieved, &count_array);
if (err != AW_NO_ERROR) {
    printf("Error on getting event sequence field\n%s\n",
        awErrorToString(err));
    return 0;
}
```

The manner in which the target array is specified depends on the sequence's type.

Function Name	Value Type
awGetBooleanSeqField	BrokerBoolean **
awGetByteSeqField	char **
awGetCharSeqField	char **
awGetDateSeqField	BrokerDate **
awGetDoubleSeqField	double **
awGetFloatSeqField	float **

Function Name	Value Type
<code>awGetIntegerSeqField</code>	<code>long **</code>
<code>awGetLongSeqField</code>	<code>BrokerLong **</code>
<code>awGetShortSeqField</code>	<code>short **</code>
<code>awGetStringSeqField</code>	<code>char ***</code>
<code>awGetUCCharSeqField</code>	<code>charUC *</code>
<code>awGetUCStringSeqField</code>	<code>charUC ***</code>

You may also use the [awGetSequenceField](#) to obtain the values and type for an event sequence field.

The [awGetSequenceFieldSize](#) can be used to obtain the number of elements in a particular sequence field.

Unicode Transform Function 8

The webMethods Broker API allows you to retrieve and set Unicode string sequence fields that use the *Unicode Transform Function 8* (UTF-8) encoding. UTF-8 allows 16-bit Unicode characters to be represented as a sequence of up to four 8-bit values. The following functions are provided:

- [awGetUCStringSeqFieldAsUTF8](#)
- [awSetUCStringSeqFieldAsUTF8](#)

Structure Data Fields

Structure data fields represent event fields with a user-defined type. A structure data field is similar in concept to a struct in the C language, because it may contain members that are simple data types, arrays, or other structures.

The following example illustrates a hypothetical event type containing a structure field.

```
Sample::StructEvent {
    string count;
    struct sample_struct {
        BrokerDate sample_date;
        int sample_array [] [];
    }
}
```

Setting Structure Fields

Two functions, `awSetStructFieldFromEvent` and `awSetStructSeqFieldFromEvents`, are provided for setting all of the values within a structure field. The first function sets the entire contents of a single structure field and the second function sets the entire contents of a sequence of structure fields.

You may also use the [awSet<type>Field](#) to set the value of a single field within a structure field by specifying the appropriate field name.

Setting a Struct Field from an Event

Since structure field may contain other fields, the [awSetStructFieldFromEvent](#) function allows you to set all of those contained values with just one function call. To use this function, simply follow these steps.

1. Create an empty event of any event type, using the [awNewBrokerEvent](#) function. Since the data you will put into this event is not likely to match a real event type definition, create this event without a Broker client so it will not be type checked.
2. Set the fields and values of the event created in step 1 so that they match the fields in the structure you want to set. Use the [awSet<type>Field](#) and [awSet<type>SeqField](#) functions.
3. Call the “[awSetStructFieldFromEvent](#)” on page 353 function, passing the event created in the above steps as the source value.
4. Envelope fields on the source event are ignored.

Setting a Struct Sequence Field from an Event

Similar in concept to [awSetStructFieldFromEvent](#), this function sets a sequence of structure fields from an array of events. Envelope fields on the source events are ignored. As with the other functions for setting sequence fields, this function may overwrite all or part of the destination structure sequence field. It may also increase the size of the target sequence, but it will never reduce the size of the sequence. To reduce the size of a sequence field, use the [awSetSequenceFieldSize](#) function.

Getting Structure Field Values

Two functions, [awGetStructFieldAsEvent](#) and [awGetStructSeqFieldAsEvents](#), are provided for obtaining all of the values from a structure field in a single function call. The first function obtains all of the values of a single structure field and the second function obtains all the values from a sequence of structure fields.

You may also use the [awGet<type>Field](#) to obtain the value of a single field within a structure field by specifying the appropriate field name.

Getting a Struct Field as an Event

Since structure fields may contain other fields, the [awGetStructFieldAsEvent](#) function allows you to obtain all of those contained values with just one function call. To use this function, simply declare a `BrokerEvent` pointer and pass its address to the [awGetStructFieldAsEvent](#) function.

As described on “[Field Type Checking](#)” on page 22, the retrieved structure is not type checked.

Getting a Struct Sequence Field

Similar in concept to `awGetStructFieldAsEvent`, this function obtains a sequence of structure fields as an array of events. Envelope fields on the target events are ignored.

As described on “[Field Type Checking](#)” on page 22, the retrieved structures are not type checked.

Envelope Fields

Many of the event envelope fields are managed for you by the `webMethods Broker` library functions. Some event fields can be set by your application and others contain values set by the Broker. You cannot set the envelope fields that are set by the Broker, but you can retrieve their values.

Note:

Attempting to retrieve an envelope field that has not been set will return a `BrokerError` with a major code set to `AW_ERROR_FIELD_NOT_FOUND`.

Envelope Field	Event Editor Type	Description
<code>age</code>	<code>long</code>	The time in seconds since the event was received by the Broker.
<code>appLastSeqn</code>	<code>int</code>	The sequence number of the last event in the sequence. Your application defines how this field is used. The conventional way to use this field is to set it to 0 if you do not know the last sequence number. By specifying the last in the sequence, the recipient knows how many events to expect.
<code>appPassword</code>	<code>unicode_string</code>	Represents a user's password.
<code>appSeqn</code>	<code>int</code>	The sequence number, set by the event's publisher. Your application defines how this field is used. The conventional way to use this field is to count upward from 1.
<code>appUsername</code>	<code>unicode_string</code>	Represents a user's name.
<code>controlLabel</code>	<code>short[]</code>	Represents the access label that a receiving client must have to receive the event. See awSetClientAutomaticControlLabel for more information.
<code>errorsTo</code>	<code>unicode_string</code>	The client ID to which the event should be forwarded if any errors are generated when this event should be sent, instead of sending to the originator of the request.

Envelope Field	Event Editor Type	Description
errorRequestsTo	unicode_string	The client ID to which a request event will be forwarded if any errors generated processing the request. If field is not set, any request event that generates an error will be discarded.
maxResults	int	The maximum number of reply events a requestor would like to receive. A value of 0 indicates that no reply or acknowledgment should be sent for this event.
replyTo	unicode_string	The client ID to which the replies to this event should be sent, instead of sending to the originator of the request.
signature	byte[]	A byte sequence that holds a digital signature.
signatureType	unicode_string	Describes the type of digital signature being used.
startResult	int	A value ≥ 0 that specifies the starting number of the event to receive. Often used in conjunction with maxResults.
tag	int	Used in the request-reply model, described in “Using Request-Reply” on page 91 , to match a request event with its corresponding reply event.
trackId	unicode_string	This field's value can be set by a publishing client application to a unique identifier for tracking purposes. This allows an event that is republished to be tracked. It also allows multiple events associated with a single logical transaction to be tracked. If not set, this field should be treated as if it contained the same value as the pubId field.
transactionId	unicode_string	This field's value can be set by a publishing client application to indicate that an event is part of a transaction. See “Transactional Client Processing with Adapters” on page 423 for more information.
transformState	unicode_string	This field allows clients that transform data to mark an event's current state. An event could be published with a transformState value of USEnglish. A receiving client could translate the event into French and publish it with a transformState value of French.

The `appSeqn` and `appLastSeqn` can be used by your publisher and subscriber applications for whatever purpose they require. One possibility is to track a sequence of events which represent a response to a single request event. Your publisher could start `appSeqn` at 1 and set `appLastSeqn` to the sequence number of the last event in the sequence. If your publisher does not know the length of the sequence when it starts publishing, then `appLastSeqn` need not be set. When your publisher is about to publish the last event of the sequence, `appLastSeqn` could be set to equal `appSeqn`. Setting both `appSeqn` and `appLastSeqn` are set to -1 indicates the event is empty.

If you want your publisher to have a continuous stream of sequence numbers, then you should use `awSetEventPublishSequenceNumber`.

Note:

The `awSetPublishSequenceNumber` function does not actually set the `pubSeqn` envelope field. Instead, it specifies the sequence number that the Broker is to use when the event is published by your Broker client.

Read-only Envelope Fields

The following table shows the envelope fields used by the Broker which your application may retrieve, but not alter.

Note:

Attempting to retrieve an envelope field that has not been set will return a `BrokerError` with a major code set to `AW_ERROR_FIELD_NOT_FOUND`.

Envelope Field	Event Editor Type	Description
<code>connectionIntegrity</code>	<code>unicode_string</code>	Indicates whether or not the received event passed over an insecure link.
<code>destId</code>	<code>unicode_string</code>	Client ID of the event's recipient. This is used only with delivered events, described on “Publishing and Delivering Events” on page 69 .
<code>enqueueTime</code>	<code>date</code>	The date and time that the Broker enqueued the event for the recipient.
<code>pubDistinguishedName</code>	<code>unicode_string</code>	The distinguished name of the Broker client that published the event using an SSL connection.
<code>pubId</code>	<code>unicode_string</code>	Client ID of the event's publisher. If the publishing client is connect to a different Broker than the recipient, the ID will be fully qualified (prefixed with the name of the publisher's Broker).
<code>pubNetAddr</code>	<code>sequence of bytes</code>	A sequence of bytes in string format that contains the IP address and port number

Envelope Field	Event Editor Type	Description
		of the event's publisher. See “The pubNetAddr Envelope Field” on page 36 for more information on this field.
pubSeqn	long	A 64-bit value representing the event's publish sequence number. The use of publish sequence numbers is described in “Using Sequence Numbers” on page 397 .
pubLabel	short[]	Set by the Broker for an event published by a client which has an access label.
recvTime	date	The date and time the event was received by the Broker.
route	sequence of structs	See “The route Envelope Field” on page 36 for more information on this field.

Your application can use the `awGet<type>Field` functions to obtain the values of an event that it has received. Be sure to use the appropriate function for the envelope field's type.

Note:

When referring to envelope fields, you must add `_env.` to each of the field names shown in the preceding table.

The following example shows how your Broker client can retrieve the `pubSeqn` field from a received event by using the `awGetLongField` function and specifying the `_env.pubSeqn` field name:

```

. . .
BrokerLong seq_number;
. . .
err = awGetLongField(event, "_env.pubSeqn", &seq_number);
if (err != AW_NO_ERROR) {
    printf("Error on get long\n%s\n", awErrorToString(err));
    return 0;
}
. . .

```

The connectionIntegrity Envelope Field

The `connectionIntegrity` envelope field is a read-only field that describes the integrity of the Broker-to-Broker connections that were used to transport the event.

Value	Meaning
empty string	At some point, the event traveled over a connection that was not encrypted.

Value	Meaning
"Export"	All the connections used to transport the event had an encryption strength of <code>AW_SSL_LEVEL_US_EXPORT</code> or greater.
"US Domestic"	The event traveled exclusively over connections with an encryption strength of <code>AW_SSL_LEVEL_US_DOMESTIC</code> .

Note:

The `connectionIntegrity` field is set by the Broker, so it does not reflect the integrity of the connection between the receiving Broker client and its Broker.

The `pubNetAddr` Envelope Field

This read-only envelope field is set by Broker and contains the IP address and port number of the event's publisher. This field supports IPv6 addresses in the string format.

Prior to the string format, the `pubNetAddr` Envelope field was six-byte long, where the first four bytes represented IPv4 address in the network byte-order, and the last two bytes represented the port in the network byte-order.

To retrieve the IP address in the old six-byte format, set the `pub-net-address-in-deprecated-format` configuration parameter to 1 in the Broker configuration file (`awbroker.cfg`). By default, `pub-net-address-in-deprecated-format = 0`, where the IP address is retrieved in the string format.

The presence of the `pubNetAddr` Envelope field is controlled by the Broker configuration file. This envelope field is disabled by default, by you can enable it using the following steps:

1. Edit the Broker configuration file and add the following line:

```
pub-net-address-in-envelope=1
```

2. Save the configuration file.
3. Restart the Broker Server.

Note:

This will enable the `pubNetAddr` envelope field for all Brokers within the Broker Server.

For complete information on the Broker configuration file, see *Administering webMethods Broker*.

The `route` Envelope Field

The `route` read-only envelope field is a sequence of structures that contain event forwarding information. This field is only set on those events which are forwarded from one Broker to another Broker. This envelope field will not be set if both the publishing and receiving clients are both connected to the same Broker. Each structure in the sequence has the format shown below.

Type	Member Name	Description
string	Broker	Name of the Broker.
date	recvTime	Time the Broker received the event from the publishing client or the other Broker.
date	enqueueTime	Time the Broker enqueued the event for the next Broker.

The values for `route[0]` will represent the originating Broker. Each time the event is forwarded to another Broker, an additional structure will be added to the `route` sequence.

For more information on Broker-to-Broker communication, see *Administering webMethods Broker*.

Specifying Field Names

Many of the functions offered by the `BrokerEvent` class require a field name parameter. How you specify a field name depends on the type of field being referenced and the function being used. The following example shows a hypothetical event type that contains several different types of data fields.

Note:

Event field names are case sensitive.

```
Sample::StructEvent {
    string title;
    float seqA[];
    int seqB[][];
    struct structA{
        BrokerDate date;
        int seqC[] [];
    }
    struct structB{
        BrokerDate time;
        struct emp {
            string name;
            string ssn;
        }
    }
    struct structC[] {
        BrokerDate date;
        int seqD[] [];
    }
}
```

The following table shows the field names you could specify to reference the various fields within the event type shown in this example.

Field Name	Description	Related BrokerEvent Functions
<code>title</code>	The <code>BrokerString</code> value.	awGetStringField

Field Name	Description	Related BrokerEvent Functions
		awSetStringField
seqA	The entire sequence of float values.	awGetSequenceField awSetSequenceField
seqA[0]	First float value in seqA	awGetFloatSeqField awSetFloatSeqField
seqA[]	Used to obtain the field's type when the number of elements in the sequence is not known.	awGetEventFieldType
seqB[0][0]	First int value in seqB.	awGetIntegerField awSetIntegerField
seqB[][]	Used to obtain the field's type when the number of elements in the sequence is not known.	awGetEventFieldType
structA	The entire structA structure.	awGetStructFieldAsEvent awSetStructFieldFromEvent
structA.date	BrokerDate value within structA.	awGetDateField awSetDateField
structA.seqC[0][0]	First int value in seqC, within structA.	awGetIntegerSeqField awSetIntegerSeqField
structA.seqC[][]	Used to obtain the field's type when the number of elements in the sequence is not known.	awGetEventFieldType
structB	The entire structB structure.	awGetStructFieldAsEvent awSetStructFieldFromEvent
structB.time	BrokerDate within structB.	awGetDateField awSetDateField
structB.emp	The structure within structB.	awGetStructFieldAsEvent awSetStructFieldFromEvent
structB.emp.name	The BrokerString within the emp structure, within structB.	awGetStringField awSetStringField

Field Name	Description	Related BrokerEvent Functions
<code>structC</code>	The entire structure sequence <code>structC</code> .	awGetStructSeqFieldAsEvents awSetStructSeqFieldFromEvents
<code>structC[0].date</code>	BrokerDate within the first structure in the <code>structC</code> sequence.	awGetDateField awSetDateField
<code>structC[].date</code>	Used to obtain the field's type when the number of elements in the sequence <code>structC</code> is not known.	awGetEventFieldType
<code>structC[0].seqD[0][0]</code>	First <code>int</code> value in <code>seqD</code> , within the first structure in the <code>structC</code> sequence.	awGetIntegerField awSetIntegerField
<code>structC[].seqD[][]</code>	Used to obtain the field's type when the number of elements in the sequences <code>structC</code> and <code>seqD</code> are not known.	awGetEventFieldType

3 Using Broker Clients

- Understanding Broker Clients 42
- Creating and Destroying Broker Clients 43
- Disconnecting and Reconnecting 46
- Connection Notification 49
- Obtaining Client State and Status 50
- Advanced Features 51
- Broker Connection Descriptors 53

Understanding Broker Clients

A client program creates one or more Broker clients in order to *publish* or retrieve events. For example, a network monitoring application might create a Broker client to publish events that represent network transmission errors. A network management application might create a Broker client to *subscribe* to these network error events. If the number of network errors events retrieved reaches a critical threshold, the management application might create a different Broker client and use it to publish statistics about the network failure.

Event-based client applications are de-coupled for one another because they generate and receive events through an entity called a *Broker Server*. When your client program creates a Broker client, it is actually establishing a connection between your application and a Broker running on the local host or some host on the network.

Client State

When a connection is established between your Broker client and a Broker, the Broker creates and maintains a *client state* that includes the following information:

- A client identifier that uniquely identifies your Broker client to a particular Broker.
- A queue where events received for your Broker client will be stored until they are retrieved.
- A list of event subscriptions for your Broker client.
- The *client group* with which your Broker client is associated.

A single client state may be shared by multiple Broker clients, as described in [“Sharing Client State” on page 55](#).

Client Groups

Every Broker client belongs to a *client group*, which provides important security features by limiting the behavior of the member Broker clients. Client groups are defined and maintained by your webMethods Broker administrator. Each client group is given a name. The restrictions on client group names are discussed in [“Parameter Naming Rules” on page 407](#).

Client groups define these important limits:

- The event types that may be delivered to group members and for which they may register subscriptions. See [“Event Subscriptions” on page 60](#) for more information.
- The event types that may be published or delivered by the group members.
- The client *life cycle*, or how long the Broker will maintain client state information for each group member.
- The client queue type, which determines how the events are stored by the Broker.
- Security information that determines which entities may create a Broker client in this group.

Client 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 your Broker client.

If the life cycle is *explicit-destroy*, the client state can only be destroyed by a system administrator or by your client application calling the `awDestroyClient` function. The *explicit-destroy* life cycle is useful for client applications that need to maintain state information even if a network or system failure occurs.

If the life cycle is *destroy-on-disconnect*, the Broker will destroy the client state whenever the connection between the Broker client and the Broker is lost. The *destroy-on-disconnect* life cycle is used by client applications that do not need to maintain state information in the event of a network or system failure.

Queue Storage Types

The client *queue storage type*, also defined by your Broker client's client group, may be either *volatile* or *guaranteed*.

Queue Type	Description
<i>volatile</i>	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.
<i>guaranteed</i>	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.

Note:

Storage types may also be defined for a particular event type. See [“Obtaining Storage Type Property” on page 115](#) for information on how client group and event type storage specifications interact.

Client Infoset

Each Broker client is allowed to store information about its state or configuration in a single client infoset. The `awGetClientInfoset` function allows you to obtain the infoset for a particular Broker client. The `awSetClientInfoset` function allows you to set the infoset for a particular Broker client. For convenience, the client infoset is treated by both of these functions as a `BrokerEvent`.

Creating and Destroying Broker Clients

Creating a Broker Client will establish a connection between your application and a Broker. This connection is identified by the following tuple:

- The name of the host where the Broker is executing.

- The IP port number assigned to the Broker.
- 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, as described in [“Sharing Connections” on page 55](#).

Your application may create a Broker client by calling the `awNewBrokerClient` function and specifying these parameters:

- The name of the host where the Broker to which you wish to connect is executing.
- 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.
- A unique client ID for identifying your Broker client. You may specify `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 `awGetClientID` function.
- 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.
- 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 connection descriptor to use for the new Broker. If you specify `NULL`, a new connection descriptor will be created for you. Connection descriptors are covered in detail in [“Broker Connection Descriptors” on page 53](#).

The example below contains an excerpt from the `publish1.c` sample application that shows the use of the `awNewBrokerClient` function. 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."

Note:

See [“Parameter Naming Rules” on page 407](#) for restrictions on Broker names.

A `NULL` client ID is specified, indicating that the Broker should generate a unique client ID. A client group named "default" is requested and the application name is set to "Publish Sample #1." A `NULL` connection descriptor is passed, indicating that the caller wishes a default connection to be established. The last parameter is used to return the `BrokerClient` reference, since the return value of the function represents the overall success or failure of the function call.

The following example illustrates how to create a Broker client:

```
#include "aweb.h"
. . .
char *Broker_host = "localhost";
char *Broker_name = NULL;
char *client_group = "default";
. . .
int main(int argc, char **argv)
```

```

{
    BrokerClient c; BrokerError err;
    . . .
    /* Create a client */
    err = awNewBrokerClient(broker_host, broker_name,
        NULL, client_group, "Publish Sample #1", NULL, &c);
    if (err != AW_NO_ERROR) {
        printf("Error on create client\n%s\n", awErrorToString(err));
        return 0;
    }
    . . .
}

```

Client Identifiers

The client identifier is a string that uniquely identifies your Broker client. The client identifier is used to:

- Identify the Broker client that published an event, described in [“Read-only Envelope Fields” on page 34](#).
- Identify the recipient of a delivered event, described in [“Delivering Events” on page 72](#).
- Reconnect a previously disconnected `BrokerClient`, described on [“Disconnecting and Reconnecting” on page 46](#).
- Allow two or more Broker clients connected to the same Broker to share the same client state, described on [“Sharing Client State” on page 55](#).

Note:

A client identifier cannot start with a '#' character, nor can it contain '/' or '@' characters. See [“Parameter Naming Rules” on page 407](#) for complete details.

Assigning Client Identifiers

It is always best to let the Broker assign a unique client identifier at the time you create your Broker client. You may then retrieve your client identifier by calling the `awGetClientID` function.

Obtaining Client Identifiers

The best way to obtain another Broker client's identifier is by retrieving the `pubId` envelope field from an event published by that client, as described on [“Obtaining the Client Identifier” on page 72](#).

Hard-coding Client Identifiers

In some cases, you may have a compelling reason to hard-code a client identifier into your application or to accept the client identifier as a command-line argument:

1. If your Broker client is designed to be disconnected and reconnected over multiple sessions and needs to preserve the client identifier, you may want to hard-code the client identifier.
2. If your application cooperates with other applications that need to know your client identifier, you may choose to hard-code the client identifier into each of the cooperating applications. If

the applications are connected to different Brokers in a multi-Broker environment, you must hard-code the fully qualified client identifier by prepending the name of the Broker to which the client is connected to the client identifier, as shown below.

Client	Unqualified Client ID	Client's Broker Name	Fully Qualified Client ID
Joe	1000	Denver	//Denver/1000
Mary	1001	Denver	//Denver/1001
Chuck	1000	Chicago	//Chicago/1000

Destroying a Broker Client

The example below contains another excerpt from the `publish1.c` sample application that shows the use of the `awDestroyClient` function. The only parameter passed to this function is the `BrokerClient` reference that was obtained by the earlier `awNewBrokerClient` function call. When a Broker client is destroyed, its event queue and all other client state information will also be destroyed.

The following example illustrates how to destroy a Broker client:

```
. . .
err = awDestroyClient(c);
if (err != AW_NO_ERROR) {
    printf("Error on client destroy\n%s\n", awErrorToString(err));
    return 0;
}
. . .
```

Using Several Broker Clients

You may find that a number of programming situations are made easier by creating several Broker clients within a single application.

- If your application has several phases of operation, you may use a separate client to represent each phase.
- Since different Broker clients can belong to distinct client groups, they can subscribe to and publish different event types. You can use this feature to divide the work your application needs to perform.
- A multi-threaded application can spawn separate threads for each Broker client, which can result in a cleaner programming model.

Disconnecting and Reconnecting

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 is 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 Broker client

You may disconnect your Broker client by calling the `awDisconnectClient` function, as shown in the example below. 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 Broker client.

```

. . .
err = awDisconnectClient(c);
if (err != AW_NO_ERROR) {
    printf("Error on client disconnect%s\n", awErrorToString(err));
    return 0;
}
. . .

```

Reconnecting a Broker Client

You can reconnect a previously disconnected Broker client that has a life cycle of *explicit-destroy* by calling the `awReconnectClient` function 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 call to `awDisconnectClient` was made.

Note: [“Automatic Reconnection” on page 54](#) describes how you can enable automatic reconnection for your Broker client in the event the connection to the Broker is lost.

The following example illustrates how to reconnect a Broker client.

```

#include "aweb.h"
. . .
int main(int argc, char **argv)
{
    char *broker_host = "localhost";
    char *broker_name = NULL;
    char *client_group = "default";
    char *client_id = NULL;
    BrokerClient c;
    BrokerError err;

    /* Create a new client */
    err = awNewBrokerClient(broker_host, broker_name,
        NULL, client_group, "Publish Sample #1", NULL, &c);
    . . .
    /* Save the client ID */

```

```
err = awGetClientID(c, &client_id);
...

/* Disconnect the client */
err = awDisconnectClient(c);
...
/* Reconnect a client with the same client_id */
err = awReconnectBrokerClient(broker_host, broker_name,
    client_id, NULL, &c);
if (err != AW_NO_ERROR) {
    printf("Error on reconnect client\n%s\n", awErrorToString(err));
    return 0;
}
. . .
```

Using the `awNewOrReconnect` Function

You may find it convenient to use the `awNewOrReconnectBrokerClient` function to create or reconnect a client when your client application is expected to be executed repeatedly. The `awNewOrReconnectBroker` function will attempt to create a new Broker client. If the client already exists and was simply disconnected, it will be reconnected.

The following example illustrates how to reconnect a Broker client using the `awNewOrReconnectBrokerClient` function:

```
#include "aweb.h"
int main(int argc, char **argv)
{
    char *broker_host = "localhost";
    char *broker_name = NULL;
    char *client_group = "default";
    char *client_id = "123";
    BrokerClient c;
    BrokerError err;
    . . .
    /* The first time this program is executed, a Broker client
     * will be created. Subsequent executions will simply reconnect
     * the Broker client.
     */
    err = awNewOrReconnectBrokerClient(broker_host, broker_name,
        NULL, client_group, "Publish Sample #1", NULL, &c);
    if (err != AW_NO_ERROR) {
        printf("Error on newOrReconnect client\n%s\n",
            awErrorToString(err));
        return 0;
    }
    . . .
    /* Do some processing */
    . . .
    /* Disconnect the client */
    err = awDisconnectClient(c);
    . . .
}
```

Connection Notification

The connection notification feature allows you to register a callback function for a particular Broker client that will be invoked if the client is disconnected from the Broker. The connection notification feature can be particularly useful if your Broker client is using the automatic reconnect feature and needs to know when it has been reconnected to the Broker.

Note:

Connection callback functions do not have a global scope. They must be registered separately for each Broker client that wishes to use this feature.

Defining a Connection Callback Function

The callback function you implement to handle connection notification must be declared using the following function prototype.

```
void (*BrokerConnectionCallback)(BrokerClient client,
int connect_status,
void *client_data);
```

<i>client</i>	The Broker client whose connection state has changed.
<i>connect_state</i>	The client's connection state. This will be set to either <code>AW_CONNECT_STATE_DISCONNECTED</code> , <code>AW_CONNECT_STATE_CONNECTED</code> , or <code>AW_CONNECT_STATE_RECONNECTED</code> .
<i>client_data</i>	Any data that your callback function needs to perform its processing. This data is defined at the time you register your callback function and may be <code>NULL</code> .

You can implement this function to recreate any needed client state that may have been lost or to provide other functions, such as message logging.

Registering the Callback Object

You use the [awRegisterClientConnectionCallback](#) function to register the callback function you wish to be invoked in the event your Broker client is disconnected or reconnected to its Broker. This function accepts three parameter.

- The Broker client for which the callback is being registered.
- The callback function being registered.
- A un-typed pointer to any data the function may need to complete its processing.

Note:

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

When the callback function is invoked, the `connect_state` parameter will be set to one of the following values shown below.

<code>connect_state</code>	Meaning
<code>AW_CONNECT_STATE_DISCONNECTED</code>	The client has been disconnected.
<code>AW_CONNECT_STATE_CONNECTED</code>	The Client connection has been re-established, because automatic reconnect was enabled.
<code>AW_CONNECT_STATE_RECONNECTED</code>	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 specifying a null callback function when you invoke the [awRegisterClientConnectionCallback](#) function.

Obtaining Client State and Status

There are a variety of functions you may use to obtain state and status information about a Broker client.

Basic Properties

To...	Use this method or function...
Obtain the version number of Broker	awGetBrokerVersionNumber method
Obtain the name of the application associated with a Broker client	awGetClientApplicationName function
Obtain the name of the host where the Broker that is associated with a Broker client is executing	awGetClientBrokerHost function
Obtain the name of the Broker to which a Broker client is connected	awGetClientBrokerName function
Obtain the IP port number of the Broker to which a Broker client is connected	awGetClientBrokerPort function
Obtain the name of the client group with which a Broker client is associated	awGetClientGroup function
Obtain the a Broker client's client ID	awGetClientId function

To...	Use this method or function...
Obtain a string that contains names of the client, client group, and Broker for a Broker client	awClientToString function
Determine if a Broker client is currently connected to a Broker	awIsClientConnected function
Determine if there are any pending events for a particular Broker client	awIsClientPending method
Determine if there are any pending events for any Broker client in use by the application	awIsPending function

Client Queue Functions

To...	Use this function...
Determine the number of events currently waiting in a Broker client's event queue	awGetClientQueueLength
Empty the event queue associated with a Broker client	awClearClientQueue

Important:

Use the [awClearClientQueue](#) function with care because it will delete events which have not yet been processed by the Broker client. If multiple Broker clients are sharing the same client state, calling this function can have far-reaching effects.

Advanced Features

The webMethods Broker C library provides several advanced functions that you may use to manipulate network connections and to obtain platform information.

Connection Settings

Use the [awGetDefaultBrokerPort](#) function to obtain the default IP port number that will be used by the [awNewBrokerClient](#) and [awReconnectBrokerClient](#) functions when connecting to a Broker.

The default port is used for non-SSL connections and has a default value of 6849. The default port number is also used to calculate the port numbers shown below.

Port number	Description
default port number minus 1	Used for SSL connections without client authentication.

Port number	Description
default port number minus 2	Used for SSL connections with client authentication.

Client Time-out Settings

Use the [awSetDefaultClientTimeout](#) function to set the default time-out for Broker requests. This function will also return the previous time-out settings.

The default connection time-out value is 30 seconds. You may find it useful to lower the connection time-out value in high-performance environments where a delay of 10 seconds probably indicates some sort of failure.

Platform Information

The platform information functions provide a way for your Broker clients to determine the version of the webMethods Broker C library which they are using as well as the operating system and hardware platform on which they are executing.

Platform information is stored in the form of key-value pairs. The following keys are registered by the webMethods Broker C library and cannot be changed. You may, however, add your own platform keys.

Key	Value
AdapterLang	"C"
AdapterLangVersion	<current library version>
OS	The name of the operating system under which the caller is executing.
Hardware	The name of the hardware platform on which the caller is executing.

To...	Use this function...
Obtain the value associated with a particular key	awGetPlatformInfo
Obtain the an array of all the currently defined platform keys.	awGetPlatformInfoKeys
Set the value associated with a particular key. If the platform key you specify does not exist, it will be added along with its value.	awSetPlatformInfo

Broker Territory and Broker-to-Broker Communication

webMethods Broker allows two or more Brokers to share information about their event type definitions and client groups. This enables communication between Broker clients connected to different Brokers. In order to share information, Brokers join a *territory*. 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*.

Use the [awGetClientTerritoryName](#) function to obtain the territory name of the Broker to which the Broker client is connected.

Broker Connection Descriptors

The attributes of the connection between a Broker client and a Broker are defined by the `BrokerConnectionDescriptor`. The connection descriptor can be used to control these connection attributes:

- Enables or disables the automatic reconnection of a Broker client in the case where the connection to the Broker is lost.
- Enables or disables the sharing of a network connection by more than one Broker client. Only Broker clients located within the same process may share a Broker connection.
- Enables or disables the sharing of client state by multiple Broker clients.
- Controls the use of the secure sockets layer (SSL) for authentication and encryption (described in [“Managing Event Types” on page 113](#)).

Note:

You must create a `BrokerConnectionDescriptor` and set its attributes before you use it to create or reconnect a Broker client. Changing the attributes of a connection descriptor will not affect the Broker client currently using the descriptor, but it will affect any subsequent uses of the descriptor.

Creating Connection Descriptors

You may create a connection descriptor by calling the [awNewBrokerConnectionDescriptor](#) function. By default, a newly created connection descriptor will have the following attributes:

- Connection sharing will be enabled.
- Client state sharing will be disabled.
- Event ordering will be set to `AW_SHARED_ORDER_BY_PUBLISHER`. See [“By-Publisher Event Ordering” on page 56](#) for a description of event ordering.
- The SSL certificate file will be set to `NULL`.

Obtaining a Client's Connection Descriptor

The [awGetClientConnectionDescriptor](#) method allows you to obtain the connection descriptor for a particular Broker client.

Copying Connection Descriptors

The [awCopyDescriptor](#) method allows you to make a copy of a connection descriptor.

Destroying Connection Descriptors

You use the [awDeleteDescriptor](#) method to destroy a connection descriptor.

Automatic Reconnection

You can use a connection description to enable automatic reconnection of a Broker client to a Broker in the event that the connection is lost. The automatic reconnection feature is disabled by default.

If your Broker client makes a request to the Broker and the connection has been lost, an attempt is made to reconnect to the Broker, just as if you had invoked the [awNewOrReconnectBrokerClient](#) function.

To...	Use this function...
Enable or disable the automatic reconnection feature	awSetDescriptorAutomaticReconnect function
Determine if this feature is currently enabled or disabled.	awGetDescriptorAutomaticReconnect method

Using Automatic Reconnection

You can use automatic reconnection, along with the *explicit-destroy* life cycle and guaranteed storage option, to provide a high degree of reliability in your application design.

Broker clients that use the automatic reconnection feature can put their invocations inside a retry loop and rely on the connection being established as needed. You may want to insert a time delay into these loops to avoid consuming too much CPU time.

Rules for Automatic Reconnection

Here are some important rules to keep in mind when using the automatic reconnection feature.

1. If the Broker disconnects from your client while the client is awaiting a reply from the Broker, a `AW_ERROR_CONNECTION_CLOSED` error is reflected to the client and no reconnection is attempted.

2. If your Broker client is using the callback model for event processing, automatic reconnection will be attempted (when necessary) at any of this points:
 - Each time the `awDispatch` function is invoked.
 - Each time `awMainLoop` function is invoked.
 - Each time an event is received by the `awThreadedCallbacks` function on a client thread that is still connected.
3. Broker clients with an *explicit-destroy* life cycle that are automatically reconnected will find their previous client state has been preserved, including previously registered subscriptions, events in the event queue, and all other client state.

Note:

For Broker clients with an *explicit-destroy* life cycle, previously retrieved events which were not acknowledged will be presented again after automatic reconnection. When a Broker client that is sharing its state with other clients is disconnected, the other clients may retrieve the unacknowledged events. You can avoid receiving duplicate events by explicitly acknowledging events using the `awAcknowledge`, `awAcknowledgeThrough`, `awGetEvents`, or `awGetEventsWithAck` functions.

4. Broker clients with a *destroy-on-disconnect* life cycle that are automatically reconnected will find that any previously registered subscriptions, events in the event queue, and all other client state will no longer exist. These Broker clients may use the connection notification feature, described on “[Connection Notification](#)” on page 49, to aid them in recreating their previous subscriptions and client state.

Sharing Connections

The webMethods Broker API improves client application performance by having all of your Broker clients that are accessing the same Broker share a single connection. If your client application has special requirements, you may override this behavior.

To...	Use this method...
Enable or disable the sharing of a particular connection by more than one Broker client	<code>awSetDescriptorConnectionShare</code>
Determine if a particular connection may be shared by more than one Broker client	<code>awGetDescriptorConnectionShare</code>

Sharing Client State

The webMethods Broker system allows Broker clients in different applications to share the same client state. Sharing client state allows several Broker clients, possibly executing 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.

One use for state sharing is to load balance event flows. If you have an application that processes request events, you might want to have several instances of the application available to process these requests.

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.

Note:

Shared state clients are not useful unless they are receiving events from more than one publisher.

Event Processing with Shared Client State

You have two options for how the Broker will present events to Broker clients that are sharing the same client state. The default event ordering is called *by-publisher* and is described on “[By-Publisher Event Ordering](#)” on page 56. You can also opt to use no ordering, as described on “[Event Processing without Ordering](#)” on page 57.

The `awGetDescriptorSharedEventOrdering` method returns the current event processing order as either `AW_SHARED_ORDER_NONE` or `AW_SHARED_ORDER_BY_PUBLISHER`.

You can use the `awSetDescriptorSharedEventOrdering` method to specify the desired event processing order.

By-Publisher Event Ordering

The Broker normally guarantees that events from a single publishing client cannot be processed out of order. This has important implications when more than one Broker client is sharing the same event queue. The following table shows an example client event queue containing events received from three different publishing clients; Broker client A, Broker client B, and Broker client C.

Publishing Client	Event Queue Position
Client A	1
Client B	2
Client A	3
Client C	4
Client B	5
Client C	6

Consider the following steps:

1. `BrokerClient X` receives the event from queue position 1 without acknowledging the event.
2. `BrokerClient Y` receives the event from queue position 2 without acknowledging the event.

- When `BrokerClient Y` then asks for another event, it is given the event from queue position 4 because the last event published by Client A has not yet been acknowledged. For more information on acknowledging events, see [“Using Sequence Numbers” on page 397](#).

Event Processing without Ordering

You can invoke the `awSetDescriptorConnectionShare` function, specifying a value of `AW_SHARED_ORDER_NONE`, to indicate that you do not want the Broker to guarantee the order of event processing. With an event ordering of `AW_SHARED_ORDER_NONE`, events will be presented to any of the Broker clients sharing the client queue *in the order in which they appear in the queue*.

Note:

You should only use an event ordering of `AW_SHARED_ORDER_NONE` if there is just one client publishing events and you wish to maximize the parallel processing of those events.

State Sharing Functions

To...	Use this function...
Determine if client state sharing is enabled or disabled for a particular connection descriptor	<code>awGetDescriptorStateShare</code>
Enable or disable the sharing of the client state associated with the connection descriptor.	<code>awSetDescriptorStateShare</code>

In addition to these functions, you may use the `awGetClientStateShareLimit` and `awSetClientStateShareLimit` to obtain and limit the number of Broker clients that may share a particular client state.

Calling Sequence for Sharing Client State

You may follow the steps below to allow multiple Broker clients to share the same client state.

- Call `awNewBrokerConnectionDescriptor` to create a descriptor and set the state sharing to 1 (`true`).
- Call `awNewBrokerClient` to create your first client, passing a client ID and the descriptor created in step 1.
- Once the Broker client is created with shared state, other Broker clients using the same Broker can use `awReconnectBrokerClient` and the client ID from step 2, to connect the Broker client. No special `BrokerConnectionDescriptor` settings are required when reconnecting.

Note:

All Broker clients wishing to share the same state must be connected to the same Broker.

- When finished with the Broker client, the application is responsible for calling `awDisconnectClient` or `awDestroyClient`.

Enabling SSL Features

See [“Managing Event Types” on page 113](#) for more information on enabling SSL features.

4 Subscribing to and Receiving Events

■ Overview	60
■ Event Subscriptions	60
■ Receiving Events in the Get-Events Model	62

Overview

This chapter describes the webMethods Broker C API for subscribing to, receiving and processing events. Reading this chapter will help you to understand:

- How to register and cancel event subscriptions.
- How subscription identifiers and tags are created and used with subscriptions.
- How to retrieve events from Broker using `awGetEvent` or `awGetEvents`.

Event Subscriptions

In order for your application to be able to receive and process events, it must first create a `BrokerClient`, as described in [“Creating and Destroying Broker Clients” on page 43](#). Then, any event that is permitted by the client group to which the `BrokerClient` belongs may be retrieved by your application.

Note:

You do not have to register any event subscriptions to receive a *delivered* event. For more information on delivering events, see [“Delivering Events” on page 72](#).

To receive published events, your application must first use the `BrokerClient` to subscribe to the event types that it wishes to receive. Event subscriptions are always made within the context of a particular `BrokerClient`.

Note:

Event types are defined with Software AG Designer, described in the *Software AG Designer Online Help*. Note that event types are known as *document types* in Designer.

A single `BrokerClient` may make multiple subscription requests, so subscriptions are distinguished by the unique combination of the event type name and an optional filter. Filters, covered in [“Managing Event Types” on page 113](#), allow you to receive only those events that contain certain data in which you are interested. If your `BrokerClient` wishes to make two subscriptions for the same event type name, a different filter must be specified for each subscription.

Limits on Subscriptions

The *client group* to which the `BrokerClient` belongs may limit the event types to which the Broker client may subscribe. See [“Client Groups” on page 42](#) for more information. The following example contains an excerpt from the `subscribe1.c` sample application that shows the use of the `awCanSubscribe` function to check for event subscription permission. The function requires these parameters:

- A `BrokerClient` handle.
- An event type name.
- A `BrokerBoolean` where subscription permission is returned.

```

. . .
BrokerClient c;
BrokerBoolean can_subscribe;
. . .
/* Check if can subscribe */
err = awCanSubscribe(c, "Sample::SimpleEvent", &can_subscribe);
if (err != AW_NO_ERROR) {
    printf("Error on check for can subscribe\n%s\n", awErrorToString(err));
    return 0;
}
if (can_subscribe == 0) {
    printf("Cannot subscribe to event Sample::SimpleEvent.\n");
    printf("Make sure it is loaded in the broker and \n");
    printf("permission is given to subscribe to it in the \n");
    printf("%s client group.\n", client_group);
    return 0;
}
. . .

```

You can also use the [awGetCanSubscribeNames](#) function to obtain the names of all the event types to which a Broker client can subscribe.

Subscribing to Events

Your application may subscribe to an event by calling the [awNewSubscription](#) function. The following example contains an excerpt from the `subscribe1.c` sample application that shows the use of the `awNewSubscription` function. The function accepts these parameters:

- A `BrokerClient` handle.
- An event type name.
- An event filter string or `NULL` if event filtering is not desired. Filters are described in [“Managing Event Types” on page 113](#).

```

. . .
BrokerClient c;
. . .
/* Make a subscription */
err = awNewSubscription(c, "Sample::SimpleEvent", NULL);
if (err != AW_NO_ERROR) {
    printf("Error on create subscription\n%s\n", awErrorToString(err));
    return 0;
}
. . .

```

Uniqueness of Subscriptions

Each subscription that your Broker client registers is considered to be unique, based on the Broker client handle, event type name, and filter that you specify. Once your Broker client registers a subscription with a particular Broker client, event type name, and filter combination, any attempts to register another subscription with the same subscription will be ignored.

Note:

The `awNewSubscription` function will not return an error if you attempt register the same subscription more than once, it will simply ignore the request.

You can use the `awDoesSubscriptionExist` function to determine if a subscription has already been registered.

Canceling a Subscription

Your application may cancel an event subscription by calling the `awCancelSubscription`. The following example contains an excerpt from the `subscribe1.c` sample application that shows the use of the `awCancelSubscription` function. The function accepts these parameters:

- A `BrokerClient` handle.
- The event type name, which may optionally contain a wildcard at the end.
- The event filter string that was specified with the original specification. `NULL` may be specified if event filtering was not specified in the original subscription. Event filters are described in [“Managing Event Types” on page 113](#).

```
. . .
BrokerClient c;
. . .
/* Cancel the subscription */
err = awCancelSubscription(c, "Sample::SimpleEvent", NULL);
if (err != AW_NO_ERROR) {
    printf("Error on canceling subscription\n%s\n", awErrorToString(err));
    return 0;
}
. . .
```

Using Wildcards

To register or cancel a subscription for multiple event types, you may use a single wildcard character `*` at the end of the event type name.

If you use wildcards in the event type name, the following restrictions apply:

- Only those event types that your client has permission to subscribe to will be included in the subscription.
- Wildcards cannot be used with any event type within the scope of `Broker::Trace` or `Broker::Activity`.
- If a filter string is specified for the subscription, it may not contain any data fields. The filter string may contain envelop fields, however.

Receiving Events in the Get-Events Model

The simplest way to retrieve events is to use the get-events model, in which your application follows these steps:

1. Create a `BrokerClient` pointer.
2. Use the `BrokerClient` to subscribe to one or more event types. If your `Broker` client only expects to receive delivered events, no subscriptions are necessary.
3. Enter a processing loop in which `awGetEvent` is called to retrieve the next event. You can use the `awInterruptGetEvents` function if you need to interrupt the a previous call to `awGetEvent` or `awGetEvents`.
4. Extract the desired fields from each received event using the functions described in [“Getting Started” on page 11](#).
5. Call `awDeleteEvent` to free the event when you are finished processing it.
6. Return to step 3 and receive the next event.
7. Call `awDestroyClient` when you are finished.

The following example contains an excerpt from the `subscribe1.c` sample application that shows the use of the `awGetEvent` function. The function accepts these parameters:

- A `BrokerClient` handle.
- The number of milliseconds to wait for an event, if none are currently available. This may be set to `AW_INFINITE` if you want to block indefinitely.
- A `BrokerEvent` pointer.

```

. . .
BrokerEvent e;
. . .
/* Loop getting events */
count = 1;
while(count <= num_to_receive) {
    err = awGetEvent(c, AW_INFINITE, &e);
    if (err != AW_NO_ERROR) {
        printf("Error on getting event\n%s\n",
            awErrorToString(err));
        return 0;
    }

    /* process the event */
    . . .

    awDeleteEvent(e);
    ++count;
}
. . .

```

Note:

The `awGetEvent` function automatically acknowledges all previously received events for the `Broker` client. It does not acknowledge the event that was just retrieved. See [“Using Sequence Numbers” on page 397](#) for more information on acknowledging events.

Getting Multiple Events

Your application may use the `awGetEvents` function to retrieve multiple events with a single function call, instead of calling `awGetEvent` to retrieve events one at a time.

The following example shows how the `subscribe1.c` sample application might be altered to use the `awGetEvents` function. The function accepts these parameters:

- A `BrokerClient` handle.
- The maximum number of events you want to be returned.
- The number of milliseconds to wait for an event, if no events are currently available. Set this to `AW_INFINITE` if you want to block indefinitely.
- A `long` pointer that will point to the number of events returned.
- An array of `BrokerEvent` pointers that will point to the events that were retrieved.

```
. . .
    int i;
    long receive_attempts = 10;
    long number_received;
    BrokerClient c;
    BrokerEvent *e;
    . . .
    /* Loop getting events */
    count = 1;
    while(count <= receive_attempts) {
        err = awGetEvents(c, MAX_EVENTS, AW_INFINITE,
            &number_received, &e);
        if (err != AW_NO_ERROR) {
            printf("Error on getting events\n%s\n",
                awErrorToString(err));
            return 0;
        }
        for(i = 0; i < number_received; i++) {
            /* process an event */
            . . .
            awDeleteEvent(e[i]);
        }
        free(e);
        ++count;
    }
. . .
```

Note:

The `awGetEvents` function automatically acknowledges all previously received events for the `Broker` client. It does not acknowledge the events that were just retrieved. See [“Managing Event Types” on page 113](#) for more information on acknowledging events.

Subscription Identifiers

The webMethods Broker library allows you to associate an arbitrary value, called a *subscription identifier*, with an event subscription. You can use subscription identifiers to quickly determine how a retrieved event is to be processed. You may also find it helpful to use subscription identifiers if your application plans to use the callback model for retrieving and processing events. The callback model is described in [“Using the Callback Model” on page 75](#).

Subscription IDs do not uniquely identify a particular subscription, so you can create different subscriptions and with the same subscription ID.

Note:

You cannot register the same subscription with more than one subscription ID. This means that if you register a particular subscription for a Broker client and assign it a subscription ID of 1, any attempt to register the same subscription with a different subscription ID will be ignored. See [“Uniqueness of Subscriptions” on page 61](#) for more information on what differentiates one subscription from another.

Specifying Subscription IDs

The following example illustrates the use of the `awNewSubscriptionWithId` function to associate two subscription identifiers with two different subscriptions. The `awNewSubscriptionWithId` function accepts these parameters:

- A `BrokerClient` handle.
- The subscription identifier.
- An event type name.
- An event filter or `NULL` if event filtering is not desired. Filters are described in [“Managing Event Types” on page 113](#).

```
BrokerClient c;
. . .
/* Make a subscription #1*/
err = awNewSubscriptionWithId(c, 1, "Sample::SimpleEvent1", NULL);
if (err != AW_NO_ERROR) {
    printf("Error on create subscription\n%s\n", awErrorToString(err));
    return 0;
}
/* Make a subscription #2*/
err = awNewSubscriptionWithId(c, 2, "Sample::SimpleEvent2", NULL);
if (err != AW_NO_ERROR) {
    printf("Error on create subscription\n%s\n", awErrorToString(err));
    return 0;
}
. . .
```

Obtaining Subscription Identifiers

The following example shows how you could use the `awGetSubscriptionIds` function to obtain the subscription identifier of a retrieved event.

Note:

If the filter string specified for two or more subscriptions have overlapping criteria, it is possible for a retrieved event to be associated with more than one subscription identifier. For more information on event filters, see [“Managing Event Types” on page 113](#).

The following example assumes that the subscriptions described in the previous example have already been made and that there is only one subscription identifier associated with any retrieved event. The `awGetSubscriptionIds` function accepts these parameters:

- A `BrokerEvent` handle.
- The address of a `long` where the subscription ID can be stored.

```
. . .
long *subscriptions;
long num_subs_returned;
. . .
while(1) {
    err = awGetEvent(c, AW_INFINITE, &e);
    if (err != AW_NO_ERROR) {
        printf("Error on getting event\n%s\n", awErrorToString(err));
        return 0;
    }
    /* get the event's subscription number */
    subscriptions = awGetSubscriptionIds(e, &num_subs_returned);
    if( subscriptions != NULL) {
        switch( subscriptions[0] ) {
            case 1:
                /* process Sample::SimpleEvent1 . . . */
                break;
            case 2:
                /* process Sample::SimpleEvent2 . . .*/
                break;
            default:
                break;
        }
    }
    awDeleteEvent(e);
}
. . .
```

Note:

Events that are delivered to your Broker client do not have subscriptions or subscription IDs. For more information on delivering events, see [“Delivering Events” on page 72](#).

Generating Subscription Identifiers

Your application may generate subscription identifiers itself or it may call one of the following `webMethods Broker` library functions to generate a subscription identifier.

The `awMakeSubId` function creates a subscription identifier for a specified Broker client. The first time it is called for a particular `BrokerClient`, it will return a value of 1. The second time it is called

for the same `BrokerClient`, it will return a value of 2, and so on. The subscription identifiers are not guaranteed to be unique if your application disconnects and reconnects the `BrokerClient`, as described in [“Disconnecting and Reconnecting” on page 46](#).

You may use the `awMakeUniqueSubId` function to create unique subscription identifiers for Broker clients that you intend to disconnect and reconnect. The `awMakeUniqueSubId` function contacts the Broker and looks up all the existing subscription identifiers in use by your application. It then assigns a subscription identifier that is guaranteed to be unique for all Broker clients that you may use. Another important feature is that after calling this function once, all subsequent calls to `awMakeSubId` will return unique subscription identifiers.

BrokerSubscription Objects

The `BrokerSubscription` object provides a convenient way to refer to and manage subscriptions by combining the following three subscription attributes into a C language structure:

- The event's subscription identifier.
- The event's name. A wildcard character "*" can be added to the end of the event type name field to represent multiple event types. See [“Using Wildcards” on page 62](#) for more information.
- An event filter or `NULL` if event filtering is not desired. Filters are described in [“Managing Event Types” on page 113](#).

`BrokerSubscription` objects are used by the following webMethods Broker library functions:

- `awCancelSubscriptionFromStruct`
- `awCancelTxSubscriptionFromStruct`
- `awGetSubscriptions`
- `awNewSubscriptionFromStruct`
- `awNewSubscriptionsFromStructs`

5 Publishing and Delivering Events

■ Overview	70
■ Publishing Events	70
■ Delivering Events	72

Overview

This chapter describes the webMethods Broker API for publishing and delivering events. Reading this chapter will help you to understand:

- How events are published.
- How events are delivered.

Publishing Events

When your client application publishes an event, the Broker places it in the event queue of each `BrokerClient` that has subscribed to that event type.

Note:

Event types (known as *document types* in Software AG Designer) are defined with Designer, described in the *Software AG Designer Online Help*.

In order for your application to be able to publish an event, it must first create a `BrokerClient`, as described on [“Creating and Destroying Broker Clients” on page 43](#). Your application may then use the `BrokerClient` to create an event, as described on [“Setting Structure Fields” on page 30](#). After setting the event fields, as described on [“Setting Structure Fields” on page 30](#), your application is ready to publish the event.

Limits on Publication

The client group to which the `BrokerClient` belongs may limit the event types the Broker client may publish. See [“Client Groups” on page 42](#) for more information. The following example contains an excerpt from the `publish1.c` sample application that shows the use of the `awCanPublish` function to check for event publication permission. The function requires these parameters:

- A `BrokerClient` handle.
- An event type name.
- A `BrokerBoolean` where publication permission may be returned.

```
. . .
BrokerClient c;
BrokerBoolean can_publish;
. . .
err = awCanPublish(c, "Sample::SimpleEvent", &can_publish);
if (err != AW_NO_ERROR) {
    printf("Error on check for can publish\n%s\n", awErrorToString(err));
    return 0;
}
if (can_subscribe == 0) {
    printf("Cannot publish event Sample::SimpleEvent.\n");
    printf("Make sure it is loaded in the broker and \n");
    printf("permission is given to publish it in the \n");
    printf("%s client group.\n", client_group);
    return 0;
}
```

```
}
. . .
```

You can also use the [awGetCanPublishNames](#) method to obtain the names of all the event types which a Broker client can publish.

Publishing an Event

After initializing the necessary event fields, your application may publish an event by calling the [awPublishEvent](#) function. The following example contains an excerpt from the `publish1.c` sample application that shows the use of the `awPublishEvent` function to publish a single event. The function accepts these parameters:

- A `BrokerClient` handle.
- A `BrokerEvent` handle.

```
. . .
BrokerClient c;
BrokerEvent e;
. . .
err = awPublishEvent(c, e);
if (err != AW_NO_ERROR) {
    printf("Error on publish\n%s\n", awErrorToString(err));
    return 0;
}
. . .
```

Publishing Multiple Events

Your application may publish several events with one call to the [awPublishEvents](#) function. The following example shows the use of the `awPublishEvents` function to publish multiple events. The function accepts these parameters:

- A `BrokerClient` handle.
- A `long` containing the number of events in the array.
- A `BrokerEvent` array.

```
. . .
BrokerClient c;
BrokerEvent e[5];
. . .
/* Create and initialize the event array */
. . .
err = awPublishEvents(c, 5, e);
if (err != AW_NO_ERROR) {
    printf("Error on publish events\n%s\n", awErrorToString(err));
    return 0;
}
. . .
```

The [awPublishEventsWithAck](#) method can be used by your Broker client to publish one or more events while, at the same time, acknowledging the receipt of one or more events. For more information on event acknowledgement, see [“Using Sequence Numbers” on page 397](#).

Delivering Events

In addition to publishing events to potentially many Broker clients, the webMethods Broker system also allows your application to deliver an event to a single Broker client, through the Broker. In order to deliver an event to a specific Broker client, your application must have the client identifier of that Broker client.

Note:

The recipient of a delivered event does not have to register a subscription for the event type being delivered. The recipient only needs to be permitted to receive the delivered event type, as specified by the client group of the receiving Broker client.

Obtaining the Client Identifier

The best way to obtain a Broker client's identifier is to extract it from an event that you have received from that Broker client, as shown in the following example. You may also hard code the client identifier of the recipient, if it is well known.

```
. . .
BrokerEvent e;
char *client_id;
. . .
err = awGetStringField(e, "_env.pubId", &client_id);
if (err != AW_NO_ERROR) {
    printf("Error on get string\n%s\n", awErrorToString(err));
    return 0;
}
```

...

Delivering an Event

The following example shows the use of the [awDeliverEvent](#) function to deliver a single event. This function accepts these parameters:

- A `BrokerClient` handle.
- A string containing the destination identifier (client identifier of the recipient).
- The `BrokerEvent` to be delivered.

Note:

An error *will not* be returned if the recipient, represented by the destination identifier, no longer exists.

```
. . .
BrokerClient c;
```

```

BrokerEvent e, event;
char *dest_id;
. . .
    /* create a client */
    . . .
    /* open any subscriptions */
    . . .
    /* receive an event */
    . . .
    /* create the event to be delivered and set the event fields */
    . . .
    /* obtain the client id of the recipient from the received event */
    err = awGetStringField( event, "_env.pubId", &dest_id);
    if (err != AW_NO_ERROR) {
        printf("Error on get tag\n%s\n", awErrorToString(err));
        return 0;
    }
    /* Deliver the event to the recipient */
    err = awDeliverEvent(c, dest_id, e);
    if (err != AW_NO_ERROR) {
        printf("Error on deliver\n%s\n", awErrorToString(err));
        return 0;
    }
    free(dest_id);
. . .

```

Delivering Multiple Events

The following example shows the use of the [awDeliverEvents](#) function to deliver multiple events. This function accepts these parameters:

- A `BrokerClient` pointer.
- A string containing the destination identifier (client identifier of the recipient).
- A long containing the number of events in the array.
- The array of events to be delivered.

Note:

An error *will not* be returned if the recipient, represented by the destination identifier, no longer exists.

```

. . .
BrokerClient c;
BrokerEvent e[5], event;
char *dest_id;
. . .
    /* create the events to be sent and set their event fields */
    . . .
    /* obtain the client id of the recipient */
    err = awGetStringField(event, "_env.pubId", &dest_id);
    if (err != AW_NO_ERROR) {
        printf("Error on get string\n%s\n", awErrorToString(err));
        return 0;
    }
    /* Deliver the events to the recipient */

```

```
err = awDeliverEvents(c, dest_id, 5, e);
if (err != AW_NO_ERROR) {
    printf("Error on deliver\n%s\n", awErrorToString(err));
    return 0;
}
free(dest_id);
. . .
```

The [awDeliverEventsWithAck](#) method can be used by your Broker client to deliver one or more events while, at the same time, acknowledging the receipt of one or more events. For more information on event acknowledgement, see [“Using Sequence Numbers” on page 397](#).

6 Using the Callback Model

■ Overview	76
■ Understanding Callbacks	76
■ General Callback Functions	78
■ Specific Callback Functions	79
■ Dispatching Callback Functions	80

Overview

This chapter describes the webMethods Broker callback API for receiving and processing events. Reading this chapter will help you to understand:

- How to register general and specific callback functions.
- How to define a callback function.
- How to retrieve and process events from the Broker using callback functions.

Understanding Callbacks

The callback model for processing event types allows your client application to register one or more callback functions to process event types for a Broker client. Unlike the get-events model, which can only process event types for one Broker client at a time, the callback model can receive any event type for any of your client application's Broker clients and then dispatch it to the appropriate callback function. If your client application creates several Broker clients, using the callback model frees your application from making separate calls to one of the webMethods Broker get-event functions for each Broker client.

Using Callbacks

Follow these steps to use the callback processing model:

1. Use `awCanSubscribe` to verify that the desired subscriptions are allowed.
2. Use one or more of the following to register the subscriptions:
 - `awNewSubscription`
 - `awNewSubscriptionWithId`
 - `awNewSubscriptionFromStruct`
 - `awNewSubscriptionsFromStructs`
3. Use `awRegisterCallback` to register a *general* callback function.
4. Next, you can optionally register any *specific* callback functions you desire, using:
 - `awRegisterCallbackForSubId`
 - `awRegisterCallbackForTag`
5. Process events using one of the following:
 - `awDispatch`
 - `awMainLoop`

- `awThreadedCallbacks`
6. Cancel all the callbacks using:
- `awCancelCallbacks`
 - `awCancelCallbackForSubId`
 - `awCancelCallbackForTag`
7. Cancel all the subscriptions using:
- `awCancelSubscription`
 - `awCancelSubscriptionFromStruct`
 - `awCancelSubscriptionsFromStructs`

Defining a Callback Function

All callback functions that you register must use the following function prototype:

```
BrokerBoolean <your_function_name_goes_here>(
    BrokerClient client,
    BrokerEvent event,
    void *client_data);
```

Value	Description
<i>client</i>	The client for which the event has been received.
<i>event</i>	The event that is being dispatched to this function.
<i>client_data</i>	A pointer to any data that you wish to be passed to this function when it is invoked.

Your function should return 1 (`true`) if its processing was successful or 0 (`false`) if a failure occurred. If 1 (`true`) is returned, the event will be acknowledged automatically. For information on acknowledging events, see [“Using Sequence Numbers” on page 397](#).

Important:

Any event that is passed to a callback function will be automatically deleted after the callback returns. Your callback function should not call `awDeleteEvent`. If you wish to save a copy of the event, your callback must use the [`awCopyEvent`](#) function.

Passing Arguments to Callback Functions

When you register your callback function, you may specify a *client_data* pointer to any user-defined data that may be necessary for the function to complete its processing. The *client_data* parameter might be used to point to state information that the callback function must access and update each time it is invoked.

Assume that you want to count the number of events that your application processes. When you register your callback function, you could use the *client_data* pointer to point to the counter variable. When the callback is invoked, it can increment the counter.

The following example illustrates how to set up an argument for a callback function.

```
BrokerBoolean sample_callback(BrokerClient c, BrokerEvent e, void *counter);
long count = 0;
. . .
int main(int argc, char **argv)
{
    BrokerClient c;
    long n;
    . . .
    /* Check if can subscribe */
    . . .
    /* Register callback */
    err = awRegisterCallbackForSubId(c, 1, sample_callback, &n);
    if (err != AW_NO_ERROR) {
        printf("Error on registering callback\n%s\n", awErrorToString(err));
        return 0;
    }
    . . .
}
```

The following example illustrates how to use the client data parameter in a callback function:

```
BrokerBoolean sample_callback(BrokerClient c, BrokerEvent e,
                             void *counter)
{
    /* increment counter */
    (*(long*)counter)++;
    /* perform rest of event processing */
    . . .
}
```

General Callback Functions

When using the callback model, you must register a *general* callback function for each Broker client by calling the [awRegisterCallback](#) function. When an event is received, if there are no specific callback functions registered which match the event's subscription identifier or tag, the general callback function will be invoked to handle the event.

You may have a single callback function process all of your application's events by making a separate `awRegisterCallback` call for each `BrokerClient` that your application uses, specifying the same callback function each time.

Depending on the complexity of your design, you may find that a single, general callback function is all that your client application requires.

Note:

You must register a general callback function before registering any specific callback function.

Using General Callbacks

The example below illustrates the use of the `awRegisterCallback` function to register a general callback function.

The following example illustrates how to process events with a general callback function.

```

. . .
BrokerBoolean sample_callback(BrokerClient c, BrokerEvent e, void *data);
. . .
/* Create a client */
. . .
/* Check if can subscribe */
. . .
/* Register a general callback */
err = awRegisterCallback(c, sample_callback, NULL);
if (err != AW_NO_ERROR) {
    printf("Error on registering callback\n%s\n", awErrorToString(err));
    return 0;
}
/* Open the subscription */
err = awNewSubscription(c, "Sample::SimpleEvent", NULL);
. . .
/* Do the main loop */
err = awMainLoop();
if (err != AW_NO_ERROR) {
    printf("Error in main loop\n%s\n", awErrorToString(err));
    return 0;
}
. . .

```

Specific Callback Functions

Depending on the complexity of your design, you may find that a single, general callback function is all that your client application requires. More complicated designs may need to make use of specific callback functions.

A *specific* callback function is only invoked to process an event with a particular subscription identifier or event tag for a particular Broker client. You may register a specific callback function by calling either the `awRegisterCallbackForSubId` or `awRegisterCallbackForTag`.

Event tag fields are part of the request-reply model, described in [“Using Request-Reply” on page 91](#).

Note:

If a received event matches the criteria for more than one callback, each callback function will be invoked to handle the event. The event will be automatically deleted after the last callback function returns.

Using Specific Callbacks

The example below illustrates the use of the `awRegisterCallbackForSubId` function to register a specific callback function.

```
. . .
BrokerBoolean general_callback(BrokerClient c, BrokerEvent e,
    void *data);
BrokerBoolean specific_callback(BrokerClient c, BrokerEvent e,
    void *data);
/* Check if can subscribe */
. . .
/* Register a general callback */
err = awRegisterCallback(c, general_callback, NULL);
if (err != AW_NO_ERROR) {
    printf("Error on registering callback\n%s\n", awErrorToString(err));
    return 0;
}
/* Register a specific callback */
err = awRegisterCallbackForSubId(c, 1, specific_callback, NULL);
if (err != AW_NO_ERROR) {
    printf("Error on registering callback\n%s\n", awErrorToString(err));
    return 0;
}
/* Open the subscription with the ID of 1 */
err = awNewSubscriptionWithId(c, 1, "Sample::SimpleEvent", NULL);
. . .
/* Do the main loop */
err = awMainLoop();
if (err != AW_NO_ERROR) {
    printf("Error in main loop\n%s\n", awErrorToString(err));
    return 0;
}
. . .
```

Dispatching Callback Functions

After registering your subscription's callback functions, your client application should then call one of the `webMethods Broker` dispatching functions to receive events and dispatch the appropriate callback function to process them. Only one of the following dispatching functions should be used.

Using `awDispatch`

The `awDispatch` function may be used to wait to receive a single event for any `Broker` client, dispatch the event to the appropriate callback function, and then return. The `subscribe2.c` and `subscribe4.c` sample applications provide examples of how to use `awDispatch`.

Using `awMainLoop`

The `awMainLoop` function is similar to `awDispatch`, except that it enters an event loop that will continue to receive and dispatch events until `awStopMainLoop` is called. The `subscribe3.c` sample application provides an example of how to use `awMainLoop`.

Using `awThreadedCallbacks`

In a multi-threaded environment, the `awThreadedCallbacks` function will spawn a thread that then invokes `awMainLoop`. This function simplifies your client application code by handling the thread creation for you.

Invoking `awThreadedCallbacks(1)` is identical to creating a thread and then invoking the `awMainLoop` method on that thread.

Invoking `awThreadedCallbacks(0)` is identical to invoking the `awStopMainLoop` method.

Event Dispatching Rules

When an event is received in the callback model, the following rules are used to dispatch the event.

1. If the received event has a tag field and the tag matches a registered callback, the received event is dispatched to that callback function.
2. If the received event has a subscription identifier, the event is dispatched once to each callback that matches the subscription identifier. If the event matches two event subscriptions with the same subscription identifier, the event will be dispatched twice to the callback function for that identifier.
3. The received event will be dispatched to the general callback function if the tag did not match any callback and:
 - a. At least one subscription identifier did not match a specific callback.
 - b. No subscription identifiers were matched because the event was delivered.

7 Transaction Semantics

■ Overview	84
■ Transactional Client Processing	84
■ Using Transaction Processing	85

Overview

This chapter describes the webMethods Broker C API for implementing applications that publish events using a transaction processing model. Reading this chapter will help you to understand transactional client processing with Broker.

For instruction on using transactional client processing with adapters, refer to [“Transactional Client Processing with Adapters”](#) on page 423.

Transactional Client Processing

Transactional client processing facilitates publish, acknowledge, or get event functions within the context of a transaction.

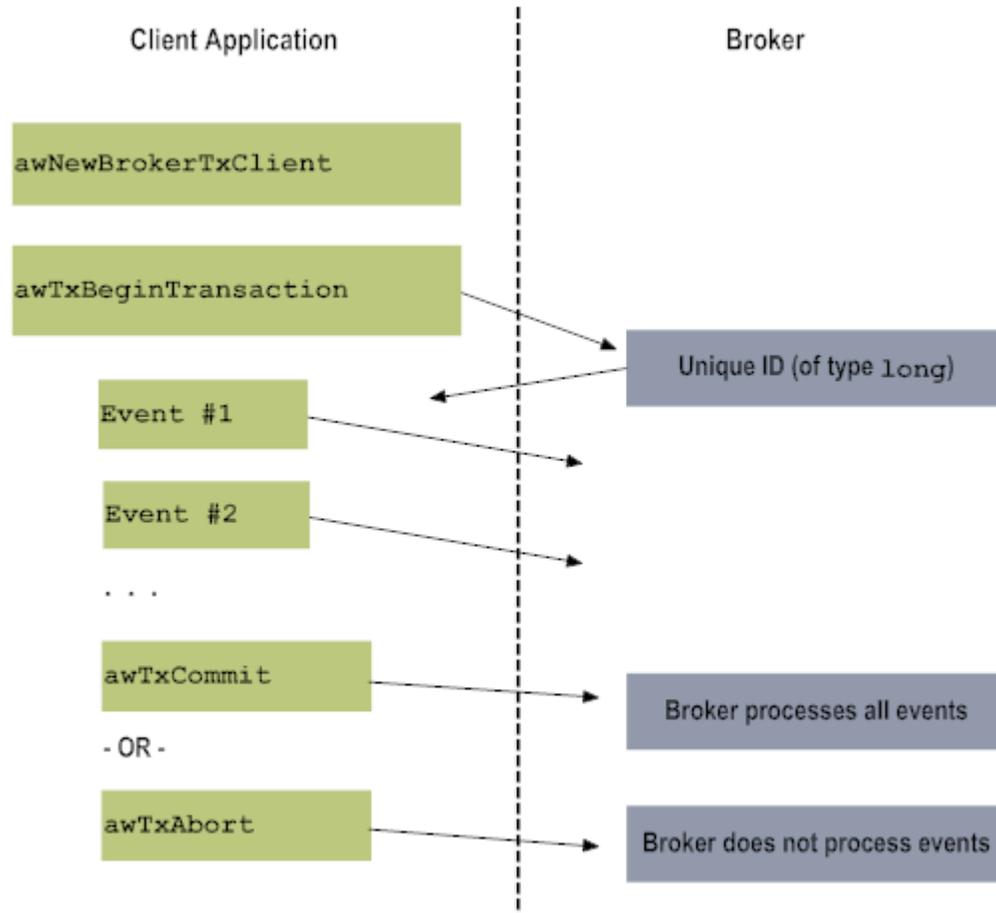
A transactional client groups the events it publishes as a single unit of work called a transaction. A transaction either completes successfully or is rolled back to some known earlier state. If a transaction fails at any state, then it is rolled back. Once all of the events that make up a transaction have been processed, your Broker client then ends the transaction. A transaction can be ended by committing the transaction or aborting the transaction.

Transaction Context Support

A Broker client initiates a transaction to publish, acknowledge, or get the events that make up the transaction, then ends the transaction by committing or aborting the events. Other features of transaction context support are:

- After an ABORT operation, all request events in the transaction are discarded and will not generate reply any events, not even error replies.
- All operations throw exceptions.
- Any request event that generates an error following a COMMIT operation will cause subsequent events to be discarded. Furthermore, any changes from prior events will be rolled back. If an error occurs, an exception (either `AWPR_ERROR_UNKNOWN_TRANSACTION`, `AWPR_ERROR_INVALID_TRANSACTION`, or `AWPR_ERROR_INVALID_CLIENT`) will be thrown.

Transaction Context Timeline



Using Transaction Processing

To use transaction processing, your application should follow these steps:

1. Obtain a third-party identifier, which will be used with all publish, deliver, get and acknowledge events that will be part of the transaction. See [“Obtaining an External Transaction ID” on page 85](#) for more information.
2. Begin the transaction. See [“Beginning a Transaction” on page 86](#) for more information.
3. Publish, deliver, acknowledge, or get the events that make up the transaction. See [“Operations Within a Transaction” on page 86](#) for more information.
4. End the transaction, which involves a COMMIT or ABORT operation. See [“Ending a Transaction” on page 89](#) for more information.

Obtaining an External Transaction ID

This is usually generated using a third-party identifier, such as a Tuxedo. Refer to the documentation of the third-party identifier for more information.

Beginning a Transaction

To begin a transaction, first instantiate a new `BrokerTxClient` object to create a transactional client. Then, use the `awBeginTransaction` function to begin the transaction. The following example illustrates the use of the `awNewBrokerTxClient` and `awTxBeginTransaction` functions to create and begin a transaction:

```
/* Create a transactional client */
err = awNewBrokerTxClient(broker_host,broker_name,"Client
    ID","default","ClientTest1",NULL,&txclient);
if (!verifyNoError(__FILE__,"awNewBrokerTxClient",__LINE__,err)) {
    return 0;
}
printf("Created broker transactional client.\n");
/* Start a transaction */
err = awTxBeginTransaction(txclient,"1",&unique_tx_id);
if (!verifyNoError(__FILE__,"awNewBrokerTxContext",__LINE__,err)) {
    return 0;
}
printf("Unique transaction id in the broker:%d\n",unique_tx_id.low);
printf("Unique transaction id in the broker:%d\n",unique_tx_id.high);
```

A unique Broker ID will be generated and returned as part of the new transaction object. This unique transaction ID can be obtained by calling the `awGetTxId` function, as illustrated in the following example:

```
err = awGetTxId(txclient, &tx_id);
if (!verifyNoError(__FILE__,"awGetTxId",__LINE__,err)) {
    return 0;
}
printf("Unique transaction id in the broker:%d\n",tx_id.low);
printf("Unique transaction id in the broker:%d\n",tx_id.high);
```

Operations Within a Transaction

The transaction object obtained in [“Beginning a Transaction” on page 86](#) is passed as a parameter into all `publish`, `deliver`, and `getEvent(s)` functions of the `BrokerTxClient` object.

Your application publishes the events that make up the transaction in the manner described in [“Using Sequence Numbers” on page 397](#).

Whether or not your application will receive an acknowledgment event or a reply event for each event it publishes depends on how the event type was defined. See *Software AG Designer Online Help* for information on defining event types.

Note:

Event types are known as *Broker document types* in Software AG Designer.

Publishing Events

When your transactional client application publishes an event, the Broker places it in the event queue of each `BrokerTxClient` that has subscribed to that event type.

Note:

For information about defining event types, see the *Software AG Designer Online Help*.

In order for your application to be able to *publish* an event, it must first create a `BrokerTxClient`, as described on “[Beginning a Transaction](#)” on page 86. Then, use the `awTxBeginTransaction` function to begin the transaction. Your application may then use the `BrokerTxClient` to create an event, as described on “[Setting Structure Fields](#)” on page 30. After setting the event fields, your application is ready to publish the event.

Determining if an Event Can be Published

By calling the `awTxCanPublish` function you can determine if an event can be published. The following example contains an excerpt that shows the use of the `awTxCanPublish` function to determine whether an event can be published:

```
. . .
err = awTxCanPublish(txclient,"APITest::Simple",&b);
if (!verifyNoError(__FILE__,"awTxCanPublish",__LINE__,err)) {
    return 0;
}
. . .
```

Publishing a Single Event

After initializing the necessary event fields, your application may publish an event within a transaction by calling the `awTxPublishEvent` function. The following example contains an excerpt that shows the use of the `awTxPublishEvent` function to publish a single event within a transaction:

```
. . .
err = awTxPublishEvent(txclient,e);
if (!verifyNoError(__FILE__,"awTxPublishEvent",__LINE__,err)) {
    return 0;
}
. . .
```

Publishing Multiple Events

Your application may publish several events with one call to the `awTxPublishEvents` function. The following example contains an excerpt that shows the use of the `awTxPublishEvents` function publishing multiple events:

```
. . .
e_array[0] = e;
e_array[1] = e;

err = awTxPublishEvents(txclient,2,e_array);
if (!verifyNoError(__FILE__,"awTxPublishEvents",__LINE__,err)) {
    return 0;
}
. . .
```

Delivering Events

In addition to publishing events to potentially many Broker clients, the webMethods Broker system also allows your application to *deliver* an event to a single Broker client, through the Broker.

Delivering a Single Event

The following example shows the use of the `awTxDeliverEvent` function to deliver a single event within a transaction.

Note:

An error *will not* be returned if the recipient, represented by the destination identifier, no longer exists.

```
. . .
err = awTxDeliverEvent(txclient,"dest",e);
if (!verifyNoError(__FILE__,"awTxDeliverEvent",__LINE__,err)) {
    return 0;
}
. . .
```

Delivering Multiple Events

The following example shows the use of the `awTxDeliverEvents` function to deliver multiple events.

Note:

An error *will not* be returned if the recipient, represented by the destination identifier, no longer exists.

```
. . .
err = awTxDeliverEvents(txclient,"dest",2,e_array);
if (!verifyNoError(__FILE__,"awDeliverEvents",__LINE__,err)) {
    return 0;
}
. . .
```

Subscribing and Getting Events

See the following sections to see the steps you use to subscribe and get events:

- [“Creating a Subscribing Client” on page 88](#)
- [“Beginning a Transaction on the Subscribing Client” on page 89](#)
- [“Getting an Event Within a Transaction” on page 89](#)

Creating a Subscribing Client

Your application may create a client that will subscribe to events transactionally. Clients are created by calling the [awNewBrokerTxClient](#) function. The following example contains an excerpt that shows the use of the [awNewBrokerTxClient](#) function in creating a subscribing client:

```

. . .
err = awNewBrokerTxClient(broker_host,broker_name,"Client
sub","default", "TransactionTest2",NULL,&txclient2);
if (!verifyNoError(__FILE__,"awNewBrokerTxClient",__LINE__,err)) {
    return 0;
}

test
printf("Created broker transactional client to subscribe events.\n");
. . .

```

Beginning a Transaction on the Subscribing Client

By calling the [awTxBeginTransaction](#) function you can begin a transaction on the subscribing client. The following example contains an excerpt that shows the use of the [awTxBeginTransaction](#) function to create a subscribing client:

```

. . .
err = awTxBeginTransaction(txclient2,"1");
if (!verifyNoError(__FILE__,"awNewBrokerTxContext",__LINE__,err)) {
    return 0;
}
. . .

```

Getting an Event Within a Transaction

By calling the [awGetTxExternalId](#) function you can begin a transaction on the subscribing client. The following example contains an excerpt that shows the use of the [awGetTxExternalId](#) function to create a subscribing client:

```

. . .
err = awTxGetEvent(txclient2, 60000, &e);
if (!verifyNoError(__FILE__,"awTxGetEvent",__LINE__,err)) {
    return 0;
}
. . .

```

Ending a Transaction

Ending a transaction can involve a COMMIT or ABORT operation.

Committing a Transaction

Use the [awTxCommit](#) function to commit and end a transaction. All outstanding events will get acknowledged once a transaction is committed. The following example contains an excerpt that shows the use of the [awTxCommit](#) function to end a transaction:

```

. . .
err = awTxCommit(txclient2);

```

```
if (!verifyNoError(__FILE__, "awTxCommit", __LINE__, err)) {  
    return 0;  
}  
. . .
```

Aborting a Transaction

Use the [awTxAbort](#) function to end a transaction. The [awTxAbort](#) function discards all work previously performed. The following example contains an excerpt that shows the use of the [awTxAbort](#) function to abort a transaction:

```
. . .  
err = awTxAbort(txclient);  
if (!verifyNoError(__FILE__, "awTxAbort", __LINE__, err)) {  
    return 0;  
}  
. . .
```

8 Using Request-Reply

■ Overview	92
■ The Request-Reply Model	92
■ The Requestor	94
■ The Server	98

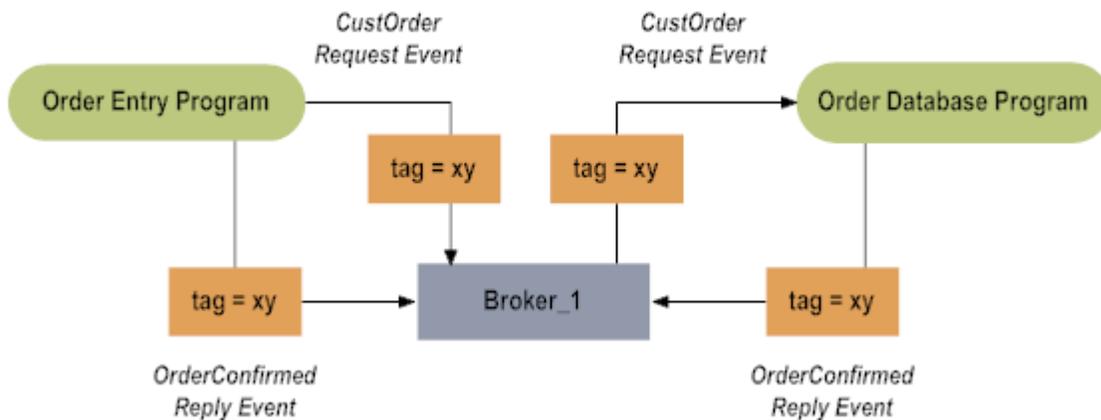
Overview

This chapter describes the webMethods Broker API for implementing applications that use the request-reply event processing model. Reading this chapter will help you to understand:

- The use of event tag fields to match request events that are sent with reply events that are received.
- The various types of reply events.
- Using `get-event`, `callback`, and the `awPublishRequestAndWait` approaches in implementing a requestor application.
- Using the reply functions in implementing a request server application.

The Request-Reply Model

You can use the request-reply model for applications that publish or deliver a request event to a server application which is expected to return a reply event. The reply event may contain data or may simply be an acknowledgment with no data.



Request Events and the Tag Field

A *tag* envelope field is set by your requestor application to identify the request event that it is sending. When a server application receives the request event and prepares the reply event, it ensures that the same tag field is set for the reply as was received on the request event. If your requestor sends several different request events, it should set each with a different tag field. When your requestor receives a reply event, it can check the tag field to determine the original request with which the reply is associated.

Getting and Setting the Tag

To...	Use this method...
Obtain the tag field from an event	awGetEventTag
Set an event's tag field	awSetEventTag

Using the trackId Field

The `trackId` envelope field can be set by a publishing client application to a unique identifier that will allow the event to be tracked. The use of this envelope field allows an event that is received and then re-published to still be tracked. This envelope field also allows you to track several events that may be associated with a single logical transaction.

Note:

If `trackId` envelope field is not set, any of the `webMethods Broker` functions for delivering reply events will automatically set it to the value contained the `pubId` envelope field.

Reply Events

The content of a reply event can vary, depending on the design of the requestor and server applications. The request event's `infoSet` defines the reply event that is expected.

A reply event may take one of the following forms:

- A null event that has no data.
- An acknowledgment that indicates the operation was successful.
- A single reply event containing application-defined data.
- A series of reply events containing application-defined data.
- An error reply.

If a series of reply events are returned, the `appSeqn` and `appLastSeqn` envelope fields may be accessed to determine the event's sequence position. See [“Envelope Fields” on page 32](#) for more information.

Determining a Reply Event's Type

The following functions are provided to help your client application determine what type of reply event has been received.

To...	Use this function...
Determine if an event is an acknowledgment reply	awIsAckReplyEvent
Determine if an event is an error reply.	awIsErrorReplyEvent

To...	Use this function...
Determine if an event is the last reply in a sequence	awIsLastReplyEvent
Determine if an event is a null reply	awIsNullReplyEvent

The Requestor

You have several processing models to choose from when designing a requestor application. These design alternatives include:

- Use `awPublishEvent` to send the request event, use `awGetEvent` to receive possible reply events, and check each event received for a matching tag.
- Create a callback function for the response event and register it with `awRegisterCallbackWithTag`, specifying the tag you will use. Use `awPublishEvent` to send the request event and then use one of the `webMethods Broker` dispatching functions to dispatch received events.
- Use `awPublishRequestAndWait` to send the request event and wait until a reply event is received.

The following example shows the code that implements the preliminary processing for requestor applications. It follows these steps:

1. Create a `BrokerClient`.
2. Check for publication permission for the request event type.
3. Check for subscription permission for the reply event type. This is done even though the reply event will be delivered because it indicates whether or not the requestor's client group will allow it to receive the reply event type.
4. Create a request `BrokerEvent`.
5. Create a tag for the request event using the `awMakeTag` function.
6. Set the request event's tag field using the `awSetEventTag` function.

```
. . .
BrokerClient c;
BrokerEvent e;
BrokerBoolean b;
int request_tag;
. . .
/* Create a client */
err = awNewBrokerClient(broker_host, broker_name, NULL
    "default", "Requestor", NULL, &c);
/* Check for errors */
. . .
/* Check if can publish */
err = awCanPublish(c, "Sample::Request", &b);
/* Check for errors */
. . .
/* Check if can subscribe */
```

```

err = awCanSubscribe(c, "Sample::Reply", &b);
/* Check for errors */
. . .
/* Create the request event */
err = awNewBrokerEvent(c, "Sample::Request", &e);
/* Check for errors */
. . .
/* Create tag and set tag field */
request_tag = awMakeTag(c);
err = awSetEventTag(e, request_tag);
/* Check for errors */
. . .

```

Using the Get-event Approach

After the preliminary processing described in [“The Requestor”](#) on page 94 has been completed, the get-event design involves these steps:

1. Use `awPublishEvent` to publish the request.
2. Enter a processing loop and receive events with `awGetEvent`.
3. Use `awGetEventTag` to check each received event for a tag that matches the request event's tag.

The following example shows how the `awGetEvent` function could be used to receive a reply event.

```

. . .
BrokerClient c;
BrokerEvent e;
BrokerBoolean b;
int request_tag;
. . .
/* Create a client */
err = awNewBrokerClient(broker_host, broker_name, NULL
    "default", "Requestor", NULL, &c);
/* Check for errors */
. . .
/* Check if can publish */
err = awCanPublish(c, "Sample::Request", &b);
/* Check for errors */
. . .
/* Check if can subscribe */
err = awCanSubscribe(c, "Sample::Reply", &b);
/* Check for errors */
. . .
/* Create the request event */
err = awNewBrokerEvent(c, "Sample::Request", &e);
/* Check for errors */
. . .
/* Create tag and set tag field */
request_tag = awMakeTag(c);
err = awSetEventTag(e, request_tag);
/* Check for errors */
. . .BrokerClient c;BrokerEvent e;BrokerBoolean done;
long received_tag, request_tag;
. . .
/* Create BrokerClient, check subscription and publish permissions,

```

```
* create request BrokerEvent, create tag, set tag field
*/
. . .
/* Publish the request */
err = awPublishEvent(c, e);
/* check for errors ... */
. . .
/* Loop getting events */
done = 0;
while(!done) {
    err = awGetEvent(c,AW_INFINITE,&e);
    if (err != AW_NO_ERROR) {
        printf(" Error on awGetEvent\n");
        return 0;
    }
    err = awGetEventTag(e,&received_tag);
    if (err != AW_NO_ERROR) {
        printf(" Error on awGetEventTag\n");
    } else if (request_tag == received_tag) {
        if (awIsNullReplyEvent(e)) {
            printf("Null reply received.\n");
        } else if (awIsAckReplyEvent(e)) {
            printf("Ack reply received.\n");
        } else if (awIsErrorReplyEvent(e)) {
            printf("Error reply received.\n");
        } else {
            /* process the event */
            . . .
        }
        done = awIsLastReplyEvent(e);
    }
    awDeleteEvent(e);
}
. . .
```

Other Reply Event Functions

1. Use [“awIsLastReplyEvent” on page 282](#) to determine if a reply event is the last in a sequence.
2. Use [“awIsNullReplyEvent” on page 283](#) to determine if a reply event is null reply event. This is indicated by the envelope fields `appSeqn` and `appLastSeqn` both being equal to -1.

Callback Functions with Tags

After the preliminary processing described in [“The Requestor” on page 94](#) has been completed, the callback design involves these steps:

1. Use `awPublishEvent` to publish the request.
2. Use `awRegisterCallback` to register a general callback.
3. Use `awRegisterCallbackForTag` to register a specific callback for the reply event with the request event's tag.

4. Use one of the callback dispatching functions, described on [“Dispatching Callback Functions” on page 80](#), to receive events and dispatch the appropriate callback function.

The following example shows how you could use the callback function to receive a reply event with a callback function:

```

. . .
BrokerBoolean test_callback1(BrokerClient c, BrokerEvent e, void *data);
int main(int argc, char **argv)
{
    BrokerClient c;
    BrokerEvent e;
    long request_tag;
    BrokerBoolean done = 0;
    . . .
    /* Publish request */
    err = awPublishEvent(c,e);
    /* check for errors ... */
    /* Register general callback */
    err = awRegisterCallback(c,test_callback1,"general");
    if (err != AW_NO_ERROR) {
        printf("Error on awRegisterCallback\n");
        return 0;
    }
    err = awRegisterCallbackForTag(c,request_tag,1,test_callback1,
        "tag matched");
    if (err != AW_NO_ERROR) {
        printf("Error on awRegisterCallback #2\n");
        return 0;
    }
    /* Dispatch loop */
    while(!done) {
        err = awDispatch(AW_INFINITE);
        if (err != AW_NO_ERROR) {
            printf(" Error on awDispatch\n");
            return 0;
        }
    }
    . . .
}
BrokerBoolean test_callback1(BrokerClient c, BrokerEvent e, void *data)
{
    char *st;
    if (awIsNullReplyEvent(e)) {
        printf("Null reply received.\n");
    } else if (awIsErrorReplyEvent(e)) {
        printf("Error reply received.\n");
    } else {
        /* process the event */
        . . .
    }
    return 1;
}

```

Using awPublishRequestAndWait

After the preliminary processing described in [“The Requestor” on page 94](#) has been completed, the callback approach involves the following steps:

1. Use `awRegisterCallback` to register a general callback.
2. Use `awPublishRequestAndWait` to publish the request and wait for the reply.

Note:

No event subscription is necessary in this model because the `awPublishRequestAndWait` function requires that the reply event be delivered, not published

The following example shows how you can `awPublishRequestAndWait`:

```
. . .
BrokerBoolean test_callback1(BrokerClient c, BrokerEvent e, void *data);
int main(int argc, char **argv)
{
    BrokerClient c;
    BrokerEvent e;
    BrokerEvent *events;
    long n;
    . . .
    /* Register general callback */
    err = awRegisterCallback(c, test_callback1, "general");
    /* check for errors */
    . . .
    err = awPublishRequestAndWait(c, e, 6000, &n, &events);
    /* check for errors */
    . . .
    /* Process received events */
    for( i = 0; i < n; i++) {
        if (awIsNullReplyEvent(events[i])) {
            printf("Null reply received.\n");
        } else if (awIsErrorReplyEvent(events[i])) {
            printf("Error reply received.\n");
        } else {
            /* process the event */
            . . .
        }
        awDeleteEvent(events[i]);
    }
    free(events);
    . . .
}
```

Delivering Request Events

You can use the `awDeliverRequestAndWait` function if you wish to send a request event to a single Broker client. This function creates a value for the `tag` envelope field using `awMakeTag` function and blocks until all reply events are received or until the requested time-out interval expires.

The Server

Server applications must be allowed to subscribe to all of the event types that are to be processed. In response to a request event, servers must also be prepared to deliver a reply event of the expected type. Generally, your server application should follow these steps:

1. Check for permission to subscribe to the request event type(s).

2. Check for the appropriate reply event publishing permissions.
3. Retrieve an event and process it.
4. Deliver the appropriate reply event(s).

Checking Subscription and Publishing Permissions

The server shown in the example below checks to see if it has permission to subscribe to the request event and to deliver the appropriate reply event. The `awCanPublish` function is used to determine if the Broker client has the necessary permissions to deliver the various reply events. In the following example, the application expects that it might have to deliver the following event types:

Event Type	Description
<code>Adapter::error</code>	Used to indicate that an exception occurred in the processing of the event.
<code>Adapter::ack</code>	Used if the infoset for <code>Sample::Request</code> specifies that a simple success or failure indication is all that is expected.
<code>Sample::Reply</code>	Used if the infoset for <code>Sample::Request</code> defines a specific event type as a response.

The following example shows how the `awCanPublish` function is used to check event publishing and subscription permissions:

```

BrokerBoolean b;
BrokerClient c;
. . .
/* Create a client */
. . .
/* Check if can publish */
err = awCanPublish(c,"Sample::Reply",&b);
if (err != AW_NO_ERROR) {
    printf("Error on awCanPublish\n");
    return 0;
}
if (b == 0) {
    printf("got false on awCanPublish for Sample::Reply\n");
    return 0;
}
err = awCanPublish(c,"Adapter::error",&b);
if (err != AW_NO_ERROR) {
    printf("Error on awCanPublish\n");
    return 0;
}
if (b == 0) {
    printf("got false on awCanPublish for Adapter::error\n");
    return 0;
}
err = awCanPublish(c,"Adapter::ack",&b);
if (err != AW_NO_ERROR) {
    printf("Error on awCanPublish\n");
    return 0;
}
}

```

```
if (b == 0) {
    printf("got false on awCanPublish for Adapter::ack\n");
    return 0;
}
/* Check if can subscribe to the request event */
err = awCanSubscribe(c,"Sample::Request",&b);
if (err != AW_NO_ERROR) {
    printf("Error on awCanSubscribe\n");
    return 0;
}
if (b == 0) {
    printf("got false on awCanSubscribe\n");
    return 0;
}
/* Subscribe to the request event */
err = awNewSubscription(c,"Sample::Request",NULL);
if (err != AW_NO_ERROR) {
    printf("Error on awNewSubscription\n");
    return 0;
}
. . .
```

Processing Request Events

Your server application has all of the usual options for receiving and processing events. It can use the `awGetEvent` function to receive events within a manually coded loop, described in [“Using Request-Reply” on page 91](#). If your server subscribes to several request event types, it can use the callback model described in [“Using the Callback Model” on page 75](#).

The server application may also associate an identifier with each of the subscriptions that it registers. When an event is received, the server application can easily determine how to process the request event. See [“Subscription Identifiers” on page 65](#) for more information.

The following example below shows an excerpt of a server application that uses the `awGetEvent` function to receive an event and determines if it is a request event:

```
. . .
/* Loop getting events */
n = 1;
while(n < count) {
    err = awGetEvent(c,AW_INFINITE,&e);
    if (err != AW_NO_ERROR) {
        printf("Error on awGetEvent\n"); return 0;
    }

    /* Check if it is a request */
    event_type_name = awGetEventTypeName(e);
    if ((event_type_name != NULL) &&
        (strcmp(event_type_name,"Sample::Request")==0)) {
        /* Process the request (see excerpts that follow) */
        . . .
    }
    free(event_type_name);
}
. . .
```

Delivering Replies

Once you have processed a request event, you can use one of several functions to deliver the appropriate type of reply event.

Note:

All of the following event delivery functions determine the destination identifier based on the `pubId` field of the original request event. None of these functions will return an error if the destination identifier refers to a Broker client that no longer exists.

Delivering Acknowledgment Replies

Your server application may deliver an acknowledgment reply if the infoset for the request event type defines that a simple success or failure indication is all that is required. When invoking the [awDeliverAckReplyEvent](#) function, you may either specify a valid publish sequence number or specify a value of zero if you do not wish to use publish sequence numbers. For more information, see [“Using Sequence Numbers” on page 397](#).

The following example illustrates how to use `awDeliverAckReplyEvent` to deliver an acknowledgment reply:

```
. . .
err = awDeliverAckReplyEvent(c,e,0);
if (err != AW_NO_ERROR) {
    printf("Error on awDeliverAckReplyEvent\n");
    return 0;
}
. . .
```

Delivering Error Replies

The server application delivers an error reply to indicate that some sort of exception occurred while attempting to process the request event, as illustrated in the following example:

```
. . .
/* Make error reply event */
err = awNewBrokerEvent(c,"Adapter::error",
    &err_event);
if (err != AW_NO_ERROR) {
    printf("Error on awNewBrokerEvent\n");
    return 0;
}
err = awDeliverErrorReplyEvent(c,e,err_event);
if (err != AW_NO_ERROR) {
    printf(" Error on awDeliverErrorReplyEvent\n");
    return 0;
}
awDeleteEvent(err_event);
. . .
```

Delivering Null Replies

The server application may deliver a null reply if the request was successfully processed and there are no data to be returned. When invoking the [awDeliverNullReplyEvent](#) function, you may either specify a valid publish sequence number or specify a value of zero if you do not wish to use publish sequence numbers. For more information, see [“Using Sequence Numbers” on page 397](#).

The following example shows the delivery of a null reply:

```
. . .
printf("Delivering null reply to requestor\n");
err = awDeliverNullReplyEvent(c,e,"Sample::Reply",0);
if (err != AW_NO_ERROR) {
    printf("Error on awDeliverNullReplyEvent\n");
    return 0;
}
. . .
```

Delivering One or More Reply Events

Your server application may deliver one or more reply events in response to a single request event by calling the [awDeliverReplyEvent](#) or [awDeliverReplyEvents](#) function. The following example shows how to send multiple reply events using [awDeliverReplyEvents](#):

```
. . .
long num;
BrokerEvent *reply_events;
. . .
/* prepare the reply events */
. . .
/* deliver the replies */
err = awDeliverReplyEvents(c,e,num, reply_events);
if (err != AW_NO_ERROR) {
    printf("Error on awDeliverReplyEvents\n");
    return 0;
}
. . .
```

Delivering Partial Replies

Your server application may need to deliver multiple reply events to satisfy a request event that it receives. If all of the reply event data cannot be read into memory or is not immediately available, the server may choose to send the reply events in batches rather than all at once.

The [awDeliverPartialReplyEvents](#) function can be used in these cases. You must set the `flag` parameter to `AW_REPLY_FLAG_START`, `AW_REPLY_FLAG_CONTINUE`, or `AW_REPLY_FLAG_END` to indicate the start, continuation, and end of the reply event sequence. If all of the replies can be sent in one batch, simply set the `flag` to `AW_REPLY_FLAG_START_AND_END`.

The following example shows the delivery of partial replies:

```
. . .
flag = AW_REPLY_FLAG_START;
while(!done) {
    /* Send the replies */
    err = awDeliverPartialReplyEvents(c,e,num,
        reply_events,flag,&token);
}
```

```
    if (err != AW_NO_ERROR) {
        printf("Error on awDeliverPartialReplyEvents\n");
        return 0;
    }
    flag = AW_REPLY_FLAG_CONTINUE;
}
flag = AW_REPLY_FLAG_END;
/* prepare the last event */

. . .
/* Send the last reply */
err = awDeliverPartialReplyEvents(c,e,num,
    reply_events,flag,&token);
if (err != AW_NO_ERROR) {
    printf("Error on awDeliverPartialReplyEvents\n");
    return 0;
}
. . .
```


9 Handling Errors

■ Overview	106
■ Using BrokerErrors	106
■ Setting the System Diagnostic Level	108

Overview

This chapter describes how to check for errors returned by webMethods Broker library functions. Reading this chapter will help you understand how to:

- Check for errors.
- Obtain error codes.
- Display the descriptive text associated with an error.
- Copy, delete, and set `BrokerError` objects.

Using BrokerErrors

Most of the webMethods Broker library functions are designed to return a `BrokerError` pointer. The webMethods Broker system has one global error per thread that is considered to be the current error. The system manages the memory associated with the current error. If your application copies the current error, remember that you are responsible for freeing the memory associated with those copies.

The `BrokerError` contains an indication of the success or failure of the function call. If the function call has returned without an error, `NULL` will be returned. The following example shows how to test for an error after an webMethods Broker library (`awNewBrokerClient`) function call:

```
BrokerError err;
. . .
err = awNewBrokerClient(broker_host, broker_name,
    NULL, client_group, "Publish Sample #1",NULL, &c);
if (err != AW_NO_ERROR) {
    /* process the error */
    . . .
```

Getting the Error Code

If your application detects an error after calling an webMethods Broker library function, you may use the `awGetErrorCode` function to obtain the error code. The possible error code values and their meanings are described in [“Managing Event Types” on page 113](#).

The following example illustrates how to obtain an error code from a `BrokerError`:

```
BrokerError err;
long errcode;
. . .
err = awNewBrokerClient(broker_host, broker_name,
    NULL, client_group, "Publish Sample #1",NULL, &c);
if (err != AW_NO_ERROR) {
    /* process the error */
    errcode = awGetErrorCode( err);
    . . .
```

Getting an Error Description

You may use the `awErrorToString` or `awErrorToCompleteString` functions to obtain a descriptive message, given a `BrokerError`. The `awErrorToString` function returns a brief message while the `awErrorToCompleteString` function returns a more detailed description. The string that is returned is intended for debugging purposes and you should not free it.

The following example illustrates how to print a brief error description:

```
BrokerError err;
. . .
err = awNewBrokerClient(broker_host, broker_name,
    NULL, client_group, "Publish Sample #1", NULL, &c);
if (err != AW_NO_ERROR) {
    printf("Error on create client\n%s\n", awErrorToString(err));
    return 0;
}
```

Other Error Functions

Several other error management functions are provided for copying, deleting, and setting `BrokerError` objects. These functions include:

- `awCopyError`
- `awDeleteError`
- `awSetCurrentError`

Preserving Errors During Recovery Processing

When your client application is handling an error, it is important to realize that any `Broker` function that you call as part of the error processing may overwrite the original error. The following example shows how to copy the original error and ensure that it is restored before returning.

If you make a copy of the global error, you must manage the memory associated with your copy. You typically want to return system managed errors.

```
. . .
BrokerError err;
BrokerError temp_err;
. . .
if (err != AW_NO_ERROR){
    temp_err = awCopyError(err);
    /* Error processing that may call other webMethods Broker functions */
    . . .
    err = temp_err;
    awSetCurrentError(err);
    return err;
}
. . .
```

Setting the System Diagnostic Level

You can use the `awSetDiagnostics` function to specify the level of error reporting that the webMethods Broker system should present to your application. You can also use the `awGetDiagnostics` function to obtain the current setting of the system's diagnostic level.

Diagnostic Level	Description
0	No error output will be produced.
1	Produces error output for major errors only. This is the default setting.
2	Produces all error output for all public functions.

10 Using BrokerDate Objects

■ Overview	110
■ Date and Time Representation	110
■ BrokerDate Functions	110

Overview

This chapter describes the webMethods Broker API for representing and manipulating date and time representations. Reading this chapter should help you understand how to create, set, and query a `BrokerDate`.

Date and Time Representation

The webMethods Broker library uses the `BrokerDate` structure, shown in the following example, along with the standard C date and time functions, to provide a simplified date and time representation for your applications.

```
typedef struct BrokerDate {
    short year;
    short month;
    short day;
    short hour;
    short min;
    short sec;
    short msec; BrokerBoolean is_date_and_time;
} BrokerDate;
```

The value of the `is_date_and_time` member is as follows:

- 0 (`false`) - only the year, month, and day are being used.
- 1 (`true`) - all the date and time fields are valid.

BrokerDate Functions

You can use the `awNewBrokerDate` to create and initialize a `BrokerDate` with the desired date and time. By default, the `is_date_and_time` member is set to 1 (`true`). If you desire an empty `BrokerDate`, use the `awNewEmptyBrokerDate` function. You are responsible for calling `awDeleteDate` when you no longer need the object.

You may use the `awNewBrokerDateSequence` to create a sequence of empty `BrokerDate` objects with their `is_date_and_time` members set to 0 (`false`). You are responsible for calling `awDeleteDate` when you no longer need the object.

Other `BrokerDate` functions include:

Method	Description
awClearDate	Clears the entire contents of this <code>BrokerDate</code>
awClearTime	Clears only the hour, minute, second, and millisecond.
awCompareDate	Determines if one <code>BrokerDate</code> greater than, equal to, or less than another <code>BrokerDate</code> .
awCopyDate	Copies one <code>BrokerDate</code> to another.

Method	Description
awDateToLocalizedString	Converts a <code>BrokerDate</code> to a string, using the current locale.
awDateToString	Converts a <code>BrokerDate</code> to a string.
awGetDateCtime	Converts a <code>BrokerDate</code> to the C <code>time_t</code> format.
awSetDate	Sets the year, month, and day.
awSetDateCtime	Sets a <code>BrokerDate</code> from a <code>time_t</code> parameter.
awSetDateTime	Sets the date and time.
awSetTime	Set only the hour, minute, second, and millisecond.

11 Managing Event Types

■ Overview	114
■ Event Type Definitions	114
■ Event Type Definition Cache	117
■ Infosets	118

Overview

This chapter describes the webMethods Broker functions for managing event types. Reading this chapter will help you understand:

- The information contained in an event type definition.
- How to obtain event type information from a Broker or from a received event.
- How to determine which Broker is managing an event type and the territory to which that Broker belongs.
- How to obtain an event type's description, storage type, and time-to-live value.
- How event type definitions are cached and how to manage the cache.
- The information contained in an infoset.
- How to access infosets.

Event Type Definitions

An event type definition contains important information about a particular event type, including the event's fields and their data type. Both the Broker and the webMethods Broker library functions use event type definitions to verify that an event's fields are set with the correct data types. The event type definitions are also used if you call the `awValidateEvent` function to validate an event that your application might have received or created.

Several functions are provided to enable your Broker client to obtain event type definitions. Once you have obtained an event type definition, you can obtain information on all of the event's fields and their associated data types. Your Broker client may need to obtain event type information to be able to dynamically handle event types whose format has changed since the client application was compiled.

Obtaining Event Type Names

In order to obtain an event type definition, you must first know the event type name.

To...	Use this function...
Obtain the event type name from an event that your Broker client has created or received	awGetEventTypeName
Obtain the names of all the event types managed by the Broker to which your Broker client is connected	awGetEventTypesNames
Obtain the names of all the event types within a particular scope	awGetScopeEventTypesNames

To...	Use this function...
Obtain the names of all the event type scope names	awGetScopeNames
Obtain the fully qualified event type name from an event type definition	awGetTypeDefTypeName
Obtain the unqualified event type name from an event type definition	awGetTypeDefBaseName
Obtain the scope name from an event type definition	awGetTypeDefScopeName

Obtaining Event Type Definitions

To:	Use this function:
Obtain a single event type definition, given an event type name	awGetEventTypeDef
Obtain multiple event type definitions from a list of event type names. You can obtain a list of event type names by using the awGetEventTypeNameNames function.	awGetEventTypeDefs
Obtain the a single event type definition for an event	awGetTypeDefFromEvent
Obtain all the event type definitions which a particular Broker client has permission to publish	awGetCanPublishTypeDefs
Obtain all the event type definitions to which a particular Broker client has permission may subscribe	awGetCanSubscribeTypeDefs

Obtaining Event Type Descriptions

Use the [awGetTypeDefDescription](#) function to obtain an event type definition's description.

Obtaining Storage Type Property

An event type's storage type property specifies the how events of its type will be stored in the Broker's incoming event queue.

Storage Type	Description
AW_STORAGE_GUARANTEED	A two-phase commit process is used to store incoming events on the Broker. This offers the lowest performance, but very little risk of losing events if a hardware, software, or network failure occurs
AW_STORAGE_VOLATILE	Local memory is used to store incoming events on the Broker. This offers higher performance along with a greater risk of losing events if a hardware, software, or network failure occurs.

Use the [awGetTypeDefStorageType](#) function to obtain the storage type for an event type.

Client groups have two storage modes; *guaranteed* and *volatile*. A client group's storage mode represents the highest storage mode allowed for any events being put into the event queue of a client belonging to that group. With one exception, the Broker will put events into a client's queue using the lesser of the client group's storage mode and the event type's storage mode, as shown below.

If the Client Group Storage Mode is...	And the Event Type Storage Mode is...	Broker will put events into a client's queue using...
Volatile	Volatile	Volatile
Volatile	Guaranteed	Volatile
Guaranteed	Volatile	Volatile
Guaranteed	Guaranteed	Guaranteed

Obtaining the Time-to-live Property

An event type's time-to-live property specifies how long an event will be available after being published before it expires and is deleted. Use the [awGetTypeDefTimeToLive](#) function to obtain an event type's time-to-live. A time-to-live value of zero means events of that type will never expire.

Obtaining Type Definitions as Strings

You can use the [awTypeDefToString](#) function to obtain a string representation of an event type definition.

Obtaining Event Field Information

Your client application can use an event type definition to obtain information about the event's fields and their associated types.

Use the [awGetTypeDefFieldNames](#) function to obtain a list of all the field names from an event type definition.

Use the [awGetTypeDefFieldType](#) function to obtain the type of a specific event field from a event type definition.

Obtaining Broker Information

Each Broker to which a Broker client may connect manages its own set of event type definitions. It may be useful for your client application to obtain information about the Broker that is managing a particular event type definition.

Use the [awGetTypeDefBrokerHost](#) function to obtain host name where the Broker managing a particular event type definition is executing.

Use the [awGetTypeDefBrokerName](#) function to obtain the name of the Broker that is managing a particular event type definition.

Use the [awGetTypeDefBrokerPort](#) function to obtain IP port number of the Broker that is managing a particular event type definition.

Broker Territory and Broker-to-Broker Communication

The webMethods Broker system allows two or more Broker Servers to share information about their event type definitions and client groups. This enables communication between Broker clients connected to different Brokers. In order to share information, Brokers join a *territory*. 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*.

Use the [awGetTypeDefTerritoryName](#) function to obtain the territory name of the Broker that is managing this event type definition.

Event Type Definition Cache

The webMethods Broker library keeps a copy of each event type definition used by your application in an *event type cache*. The cache improves the performance of the event field type checking process because the Broker managing the event type does not have to be contacted each time the event type definition is accessed.

The operation of the event type definition cache is usually transparent to your application. Problems may arise if Software AG Designer is used to alter a Broker's event definitions while your application is executing. In these cases, the locally cached event type definitions will no longer be synchronized with the Broker's type definitions.

Notification of Event Type Definition Changes

If your application needs to know when a Broker's event type definitions have changed, it may subscribe to the event type `Adapter::refresh`. Whenever this event type is received, your application will know the event type definition cache needs to be synchronized with the Broker.

Use the [awFlushTypeDefCache](#) function to cause the cache contents to be refreshed. This is an advanced function and its use is not required in most situations.

Locking the Type Definition Cache

Multi-threaded applications may run into problems if one thread is accessing event type definition information and another thread calls the `awFlushTypeDefCache` function. In these situations, threads that need to access the type definition cache should lock the cache before using the following functions:

- `awGetEventTypeName`
- `awGetEventTypeDefNames`
- `awGetEventTypeDef`
- `awGetEventTypeDefs`
- `awGetTypeDefFieldNames`
- `awGetTypeDefFieldType`
- `awGetTypeDefBrokerHost`
- `awGetTypeDefBrokerName`
- `awGetTypeDefBrokerPort`

Use the `awLockTypeDefCache` function to lock the local type definition cache and prevent it from being altered or flushed.

Once a thread has finished accessing event type definition information, it should unlock the cache using the `awUnlockTypeDefCache` function.

Infosets

Each event type has a set of one or more infosets that describe configuration information for the event type.

An event type's *public* infoset defines the semantics of the event. In particular, the public infoset defines the event type of a reply that may be expected for an event.

Note:

While infosets are neither events nor event types, the functions for obtaining them return the infoset as a `BrokerEvent` for convenience.

To...**Use this function...**

Obtain the names of all the infosets defined for a particular event type name `awGetEventTypeInfoNames`

Obtain a single infoset, given an event type name and the infoset name `awGetEventTypeInfo`

To...	Use this function...
Obtain a list of infosets, given an event type name and a list of infoset names	awGetEventTypesInfosets

12 Configuring Broker Client Security

■ Overview	122
■ Working with Basic Authentication	122
■ Working with Secure Socket Layer (SSL)	124

Overview

This chapter explains how to configure security for webMethods Broker clients by using the Broker C API. Information is provided that explains how to:

- Authenticate Broker clients using basic authentication
- Authenticate Broker clients using secure socket layer (SSL)
- Manage SSL certificates for C client applications
- Enable encryption

The Broker security model provides the following forms of protection for your event-based Broker C applications:

- **User authentication** to verify the identity of a Broker C client to the Broker Server. The two-way SSL authentication verifies the identity of a Broker Server to a Broker C client attempting to make a connection *and* that of the Broker C client to the Broker Server. Before making a connection, the basic authentication identity or the SSL identity must be assigned.
- **User authorization** for Broker objects protected by Access Control Lists (ACLs). Only clients whose basic authentication or SSL identities are specified in a Broker object's ACL may connect to that object. This type of security protects confidential data from access by unauthorized users.
- **Encryption** of the data traffic between a Broker client and the Broker Server, to protect sensitive data. Typically, you encrypt the data traffic when working with highly sensitive data, or to protect data of a confidential nature that passes across a public network. Encryption is supported for SSL enabled client. A basic authentication enabled client can also use the encryption only if the connection is SSL enabled.

Working with Basic Authentication

Using the basic authentication mechanism, a C client application can provide the user name and password credentials while requesting access to Broker Server. For more information about basic authentication, please see the *Administering webMethods Broker* guide.

Configuring the Broker C Client for Basic Authentication

This section describes how to use the Broker C client API to configure and manage basic authentication. The Broker C client API contains the functions to get and to set basic authentication related settings in BrokerConnectionDescriptor.

Configuring Basic Authentication by using the Broker Connection Descriptor

Use the `awSetDescriptorAuthInfo` function to enable basic authentication for your C client, prior to creating a Broker client. When a BrokerConnectionDescriptor is created, the authentication information

will be set to null by default. Therefore, you must use the `awSetDescriptorAuthInfo` function before creating a Broker client if you want to enable basic authentication.

Function Signature	Description
--------------------	-------------

```
BrokerError
awSetDescriptorAuthInfo(BrokerConnectionDescriptor desc, const
char *auth_username, const
char *auth_password);
```

Sets the basic authentication information, that is user name and password on the specified Broker connection descriptor.

- *desc* is the Broker connection descriptor on which you want to set the basic authentication user name and password.
- *auth_username* is the user name for basic authentication.
- *auth_password* is the password for the basic authentication user.

```
BrokerError
awGetDescriptorAuthUserName(BrokerConnectionDescriptor desc, char
**user_name);
```

Returns the user name of the basic authentication user available in the specified Broker connection descriptor.

- *desc* is the Broker connection descriptor from which you want to get the basic authentication user name.
- *user_name* is the basic authentication user name.

The following code sample shows how to configure the basic authentication settings for Broker using the `awSetDescriptorAuthInfo` function, passing in the arguments described in the table above.

```
char *broker_host = "localhost";
char *broker_name = NULL;
char *basic_auth_username = "";
char *basic_auth_password = "";
BrokerConnectionDescriptor br_cdesc;
BrokerClient client;
BrokerError err;

/* Create a Broker connection descriptor*/
    br_cdesc = awNewBrokerConnectionDescriptor();

/* Set the basic authentication user name and password in the
 * Broker connection descriptor
 * br_cdesc - The Broker connection descriptor on which
 * you want to set the basic authentication user name and password.
 * basic_auth_username - The basic authentication user name to be set
 * on the br_cdesc.
 * basic_auth_password - The basic authentication password to be set
 * on the br_cdesc.
 */
```

```

err = awSetDescriptorAuthInfo(br_cdesc, basic_auth_username,
                             basic_auth_password);

/* Create a new Broker client using the basic authentication settings
 * defined in the Broker connection descriptor
 * broker_host - The name of the host running the Broker that the
 * new Broker client will use.
 * broker_name - The name of the Broker to which the new Broker client will
 * be connected.
 * client_id - A string containing a unique identifier for the
 * new Broker client to be used when disconnecting or reconnecting.
 * client_group - The name of the client group for the new Broker client.
 * app_name - The name of the application that will contain the
 * new Broker client.
 * br_cdesc - The Broker connection descriptor to be used by the
 * new Broker client.
 * client - A pointer to the newly created BrokerClient.
 */

err = awNewBrokerClient(broker_host, broker_name, client_id,
                       client_group, app_name, br_cdesc, &client);

```

Working with Secure Socket Layer (SSL)

The SSL support in webMethods Broker provides different levels of security for the communication between a Broker client and the Broker to which it is connected. This section describes how to use the BrokerConnectionDescriptor functions to control the use of SSL.

Note:

In order for your Broker client to use SSL, your Broker must first be configured to use SSL. See *Administering webMethods Broker* for information on configuring your Broker.

Enabling SSL Authentication and Encryption

To enable the use of SSL by your Broker client, your application needs access to a certificate file, the password for that file, and a truststore file. You use this information with the [awSetDescriptorSSLCertificate](#) method to enable SSL security, prior to creating or reconnecting a Broker client.

When a BrokerConnectionDescriptor is created, the certificate file will be set to NULL by default. Therefore, you must use the [awSetDescriptorSSLCertificate](#) function before creating or reconnecting a Broker client if you want to enable SSL security.

The [awSetDescriptorSSLEncrypted](#) method allows you to control whether or not data traffic will be encrypted when SSL is enabled. When a BrokerConnectionDescriptor is created, the encrypt flag will be set to 1 (true) by default. Therefore, you must use the [awSetDescriptorSSLEncrypted](#) function before creating or reconnecting a Broker client if you wish to disable data encryption.

With these two functions, you have several options for configuring SSL:

1. Disable SSL entirely (the default).

2. Enable server only authentication with encryption of data traffic.
3. Enable server only authentication without encryption.
4. Enable both server and client authentication with encryption.
5. Enable both server and client authentication without encryption.

Server Authentication with Encryption

To use server authentication with data encryption (One-way SSL), invoke the `awSetDescriptorSSLCertificate` function with a NULL certificate file and certificate file password, and a valid truststore file. Then create or reconnect the Broker client, as shown in the following example:

```

. . .
static char *certfile = "NULL"; //NULL certificate file
static char *password = "NULL"; //NULL certificate file password
static char *trustfile = "trustfilepath";

BrokerConnectionDescriptor desc;
/* Create descriptor */
desc = awNewBrokerConnectionDescriptor();
if (desc == NULL) {
    printf("Error on create descriptor\n");
    return 0;
}

/* Enable Server authentication only */
err = awSetDescriptorSSLCertificate(desc, certfile, password, trustfile);
if (err != AW_NO_ERROR) {
    printf("Error on setting SSL cert\n%s\n",
        awErrorToCompleteString(err));
    return 0;
}
/* Create a client */
. . .

```

Server Authentication without Encryption

To use server authentication without data encryption (One-way SSL without encryption), follow the same procedure as in the previous example, but invoke the `awSetDescriptorSSLEncrypted` function before creating or reconnecting the Broker client, as shown in the following example:

```

. . .
static char *certfile = "NULL"; //NULL certificate file
static char *password = "NULL"; //NULL certificate file password
static char *trustfile = "trustfilepath";

BrokerConnectionDescriptor desc;
/* Create descriptor */
desc = awNewBrokerConnectionDescriptor();
if (desc == NULL) {
    printf("Error on create descriptor\n");
    return 0;
}

```

```

/* Enable Server authentication only */
err = awSetDescriptorSSLCertificate(desc, certfile, password, trustfile);
if (err != AW_NO_ERROR) {
    printf("Error on setting SSL cert\n%s\n",
        awErrorToCompleteString(err));
    return 0;
}
/* Disable encryption */
err = awSetDescriptorSSLEncrypted(desc, FALSE);
if (err != AW_NO_ERROR) {
    printf("Error on setting SSL encrypt flag\n%s\n",
        awErrorToCompleteString(err));
    return 0;
}
/* Create a client */
. . .

```

Server and Client-side Authentication with Encryption

To use both server and client authentication with data encryption, invoke the `awSetDescriptorSSLCertificate` function with a valid certificate file, certificate file password, and a valid truststore file, as shown in the following example:

```

. . .
static char *certfile = "certfile";
static char *password = "mypassword";
static char *trustfile = "trustfile";

BrokerConnectionDescriptor desc;
/* Create descriptor */
desc = awNewBrokerConnectionDescriptor();
if (desc == NULL) {
    printf("Error on create descriptor\n");
    return 0;
}

/* Enable both Server and client authentication*/
err = awSetDescriptorSSLCertificate(desc, certfile, password, trustfile);
if (err != AW_NO_ERROR) {
    printf("Error on setting SSL
    cert\n%s\n", awErrorToCompleteString(err));
    return 0;
}
/* Create a client */
. . .
}

```

Server and Client Authentication without Encryption

To use both server and client authentication without data encryption, follow the same procedure as in the previous example, but invoke the `awSetDescriptorSSLEncrypted` function before creating or reconnecting the Broker client, as shown in the following example:

```

. . .
static char *certfile = "certfile";
static char *password = "mypassword";
static char *trustfile = "trustfile";

```

```

BrokerConnectionDescriptor desc;
/* Create descriptor */
desc = awNewBrokerConnectionDescriptor();
if (desc == NULL) {
    printf("Error on create descriptor\n");
    return 0;
}

/* Enable SSL encrypton and authentication */
err = awSetDescriptorSSLCertificate(desc,certfile,password,trustfile);
if (err != AW_NO_ERROR) {
    printf("Error on setting SSL
cert\n%s\n",awErrorToCompleteString(err));
    return 0;
}

/* Disable encryption */
err = awSetDescriptorSSLEncrypted(desc,FALSE);
if (err != AW_NO_ERROR) {
    printf("Error on setting SSL encrypt flag\n%s\n",
awErrorToCompleteString(err));
    return 0;
}

/* Create a client */
. . .
}

```

Retrieving a Broker's Certificate

You may use the [awGetBrokerSSLCertificate](#) method to obtain the certificate of the Broker to which your Broker client is connected.

Retrieving the SSL Settings for a Descriptor

You may use the [awGetDescriptorSSLCertificate](#) method to obtain the certificate file and truststore file being used for a particular `BrokerConnectionDescriptor`.

You may use the [awGetDescriptorSSLEncrypted](#) method to determine if data encryption is enabled for a particular `BrokerConnectionDescriptor`.

Retrieving Information from a Certificate File

You may use the [awGetSSLCertificate](#) method to obtain the certificate from a certificate file for a given password.

Obtaining the SSL Encryption Strength

You may use the [awGetSSLEncryptionLevel](#) method to obtain the level of encryption available to your client application.

Note:

US federal regulations restrict the encryption key size in software that is to be exported. Software that is used domestically may use larger encryption key sizes.

Broker Ports used for SSL

For information on how the SSL port numbers are calculated using the default port number, see [“Connection Settings” on page 51](#).

Converting Certificates to Strings

You may use the `awSSLCertificateToIndentedString` or the `awSSLCertificateToString` to obtain a string containing the contents of an SSL certificate.

Certificate Files

To configure SSL for a Broker C client, you must first store its SSL client certificate information (public certificate/private key pair) in a keystore file, and the trusted root (public certificate of the client certificate issuer, or CA) in a truststore file. These certificates are used to create the Broker C client's SSL *identity*.

For the client to make an SSL connection to the Broker Server, you must also have assigned SSL identities to the Broker Server. Once the SSL certificate for the Broker Server is configured, you can create an SSL connection between the Broker C client and its Broker Server.

Detailed information about creating and managing keystores and truststores for the Broker Server and the Broker admin component is provided in *Administering webMethods Broker*.

13 Using Event Filters

■ Overview	130
■ Filter Strings	130
■ Using Filters with Subscriptions	141
■ Using BrokerFilters	142

Overview

This chapter describes the webMethods Broker event filtering facilities, which allow both the Broker and your Broker clients to quickly determine if an event's contents meet specified criteria. Reading this chapter will help you understand:

- How to specify a filter string.
- How to use a filter string with an event subscription.
- How to use a `BrokerFilter` within your client application, to match events with filter criteria.

Filter Strings

A filter string specifies criteria for the contents of an event. When you specify a filter string with an event subscription, the Broker uses the filter string to determine which events match your criteria. The Broker will place in your Broker client's event queue just the events that match the filter string.

Filter strings may do any combination of the following:

- Refer to the content of event fields by using *field identifiers*.
- Compare event field contents against constants or computed values.
- Combine event field comparisons using the boolean operators *and*, *or* and *not*.
- Perform arithmetic operations on event fields and constants.
- Contain regular expressions.
- Contain string and arithmetic constants.
- Contain a *hint* that specifies how events should be processed.

Specifying Filter Strings

Consider an event that contains a person's age and the state in which they live. The first event field might have the name *age* and the second might have the field name *state*. The following filter string would match only those events whose *age* field was greater than 65 and whose *state* field was equal to "FL."

```
age > 65 and state = "FL"
```

In this example filter string, *age* and *state* represent event fields. This filter also contains an arithmetic constant **65** and a string constant "FL". The boolean operator *and* is used to combine the field criterion for *age* and *state*.

Other example filter specifications:

```
debt > salary*0.50  
packaging = "portable" and price > 5000  
answer = 'Y' or answer = 'y'
```

```
(answer = 'Y') or (answer = 'y')
```

A filter string may also make use of the filter functions described on [“Filter Functions” on page 136](#).

Locale Considerations

The current locale for your application is defined by your operating system, and allows your application to adapt itself to different languages, character sets, time zones, date formats, and monetary conventions.

A filter string is parsed using the C language locale, regardless of your actual locale settings. This means that floating point constants must use a period (.) to represent the radix, just as you would in the C language.

Data within a filter string is handled using your current locale setting. This means that string conversion and comparisons will be affected by your current locale settings.

Important:

The Broker uses a filter string to determine which events match your criteria and that evaluation occurs using the Broker's locale, not your application's locale. This may cause unexpected or unintended event filtering.

Unicode Considerations

Unicode characters and strings may be freely mixed with and compared to ANSI characters and strings. ANSI characters and strings are automatically promoted to Unicode characters or strings, when necessary, without the need for explicit casting.

Unicode string comparison on a Windows platform is performed in a locale-sensitive manner.

Important:

On Unix platforms, Unicode strings are not collated in a locale-specific manner unless both strings contain characters less than `\u00FF`, allowing them to be downcast to ANSI strings.

Filter Rules

Filter strings must adhere to these rules:

1. Field names may be fully qualified, such as:

```
struct_field.seq_field[2]
```

2. A character constant is a single character surrounded by single quotes, such as 'A'.
3. A string constant is zero or more characters surrounded by double quotes, such as "account".
4. If a character or string constant contains a single or double quote, precede the quote with a backslash. For example, '\\".
5. Parentheses may be used to control the order of operator precedence, described in [“Operator Precedence” on page 134](#).

Field Names

When referring to a structure or sequence field within a filter string, you may use a fully qualified field name. Consider the event type definition shown in the following example that contains a structure field:

```
Sample:StructEvent {
    string count;
    int scores[];
    struct sample_struct {
        BrokerDate sample_date;
        int sample_array [];
    }
}
```

To refer the third integer in the sequence field `scores`, you would use the field name `scores[2]` in your filter string.

To refer to the field `sample_date` within the struct field `sample_struct`, you would use the field name `sample_struct.sample_date` in your filter string.

To refer to the first integer in the sequence field `sample_array`, you would use the field name `sample_struct.sample_array[0]` in your filter string.

Filter Operators

The following tables contain the various operators that you can use to create filters.

Logical Filter Operators

Operator	Description
!	Not
not	
&&	And
and	
	Or
or	

Comparison Filter Operators

Operator	Description
<	Less than

Operator	Description
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
==	Equal to
!=	Not equal to

Arithmetic Filter Operators

Operator	Description
-	Unary minus
*	Multiplication
/	Division
%	Modulus Division
-	Subtraction
+	Addition

Note:

Implicit type conversion occurs when operands in an arithmetic operation have different types. The operands are converted to a larger value before the comparison occurs. Type `char` is considered numeric, but `boolean` is not.

Bitwise Operators

Operator	Description
~	Bitwise compliment
<<	Shift left
>>	Shift right
&	Logical and
	Logical or
^	Logical exclusive or

String Operators

Operator	Description
+	Concatenation
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
==	Equal to
!=	Not equal to

Operator Precedence

The following table describes the precedence and order of evaluation of all of the filter operators. Operators appearing on the same line have the same precedence.

Operator	Description	Order of evaluation
()	Function	left to right
not	logical not	right to left
!	logical not	
~	bitwise not	
-	minus	
*	multiplication	left to right
/	division	
%	modulus division	
-	subtract	left to right
+	add	
<<	left shift	left to right
>>	right shift	
<	less that	left to right

Operator	Description	Order of evaluation
<=	less that or equal to	
>	greater than	
>=	greater than or equal to	
=	equal	left to right
==	equal	
!=	not equal	
&	bitwise and	left to right
^	bitwise exclusive or	left to right
	bitwise or	left to right
and	logical and	left to right
&&		
or	logical or	left to right

Using Regular Expressions

Regular expressions allow you to specify a complex pattern that is to be matched with an input string. The [regexpMatch](#) filter function accepts a regular expression as an argument.

A regular expression is made up of one or more basic components called *atoms*. An atom is used to match a single character in the input string or it can represent multiple occurrences of one or more characters.

If each atom in a regular expression matches a corresponding element in the input string, the expression is said to match the input string.

Character(s)	Matches
.	Any single character in the input string.
^	A null string at the start of the input string.
\$	A null string at the end of the input string.
\x	The character x. A special character that you wish to match in the input string may be escaped in this manner, such as \^ or \\$.
[chars]	Any single character from <i>chars</i> .

Character(s)	Matches
	<p>Specifying <code>[a-z]</code> will match any input string containing all lowercase characters. Specifying <code>[0-9a-fA-F]</code> will match any input string containing hexadecimal digits.</p> <p>If a <code>]</code> character appears as the first characters in <i>chars</i>, it will be treated literally instead of as a terminator.</p>
<code>[^chars]</code>	Any single character that is not contained in <i>chars</i> .
<code>(regexp)</code>	Any input string that matches <i>regexp</i> . Parentheses can be used to create complex regular expressions.
<code>*</code>	A sequence of 0 or more of the preceding atom.
<code>+</code>	A sequence of 1 or more of the preceding atom.
<code>?</code>	Either a null string or the preceding atom.
<code>regexp1 regexp2</code>	Anything that matches either <i>regexp1</i> or <i>regexp2</i> . Note that the <code> </code> delimiter cannot be preceded by or followed by a blank.

Using Hints

You may include a hint in you subscription by adding a string with the following format.

```
"{hint: HintName=Value}"
```

The following table shows the supported `HintNames` and their associated values.

HintName and Value	Description
<code>IncludeDeliver=true</code>	Causes events that are delivered to a Broker client to match a subscription, along with events that are published.
<code>LocalOnly=true</code>	In a multi-Broker environment, this causes a subscription to match only those events that originate from the Broker to which this Broker client is connected. Events originating from a different Broker are excluded from the subscription.

Filter Functions

Several functions are provided by the `webMethods Broker` library that you may use within filter strings. These functions allow you to perform complex string operations, regular expression matches, and string to numeric conversions on event field values. Some examples of the use of functions within filter strings are shown below.

```
charAt(my_string_field,5) = 'A'
charAt("StringConstant",my_int_field) = 'n'
```

```
startsWith(toUpperCase(myfield)) = "FIRSTWORD"
toString(my_long_field) = "42"
toLong(my_string_field) = 42
```

charAt

```
char charAt(
    string s,
    int index)
```

- s* The name of an event field containing a string.
- index* The index of the character within the string field that is to be returned. This index is zero-based, so the first character has an index of 0.

Returns the character with the specified *index* from the event field with the name specified by *s*.

contains

```
boolean contains(
    string s,
    string substr)
```

- s* The name of an event field containing a string.
- substr* The string that is being searched for.

Returns true if the string *substr* occurs within the event field with the name specified by *s*, otherwise false is returned.

date

```
date date(
    int year,
    int month,
    int day)
```

- year* The year.
- month* The month.
- day* The day.

Creates a date value with the specified *day*, *month*, and *year*.

date

```
date date(
    int year,
```

```
int month,  
int day,  
int hour,  
int minute,  
int second,  
int millisecond)
```

<i>year</i>	The year.
<i>month</i>	The month.
<i>day</i>	The day.
<i>hour</i>	The hour.
<i>minute</i>	The minute.
<i>second</i>	The second.
<i>millisecond</i>	The millisecond.

Creates a date value with the specified *millisecond*, *second*, *minute*, *hour*, *day*, *month*, and *year*.

endsWith

```
boolean endsWith(  
    string s1,  
    string s2)
```

<i>s1</i>	The name of an event field containing a string.
<i>s2</i>	The string being searched for.

Returns true if the event field with the name specified by *s1* ends with the string *s2*, otherwise false is returned.

regexMatch

```
boolean regexMatch(  
    string s,  
    string regexp)
```

<i>s</i>	The name of an event field containing a string.
<i>regexp</i>	A string containing a UNIX regular expression.

Returns true if the event field with the name specified by *s* matches the UNIX regular expression *regexp*, otherwise false is returned.

Important:

This function does not support the X/Open regular expression specification. Unicode characters and strings are also not supported by this function.

startsWith

```
boolean startsWith(
    string s1,
    string s2)
```

s1 The name of an event field containing a string.

s2 The string being searched for.

Returns true if the event field with the name specified by *s* begins with the string *s2*, otherwise false is returned.

substring

```
string substring(
    string s,
    int index1,
    int index2)
```

s The event field from which the substring is to be extracted.

index1 The index of the character within the string field where extraction is to begin. This index is zero-based, so the first character has an index of 0.

index2 The index of the character within the string field where extraction is to end. This index is exclusive.

Returns a substring from the event field with the name specified by *s*, beginning with the character at *index1* and ending at the character at *index2*.

toDouble

```
double toDouble(
    <type> field)
```

field The name of an event field to be converted.

Returns a `double` containing the event field *field*. Any supported field type, except for dates, structures, and sequences, may be converted.

Note:

The locale-specific radix character will be recognized when converting a floating point number.

toInt

```
int toInt(  
    <type> field)
```

field The name of an event field to be converted.

Returns an `int` containing the event field *field*. Any supported field type, except for dates, structures, and sequences, may be converted.

toLong

```
long toLong(  
    <type> field)
```

field The name of an event field to be converted.

Returns a `long` containing the event field *field*. Any supported field type, except for dates, structures, and sequences, may be converted.

toLowerCase

```
string toLowerCase(string s)
```

s The name of an event field containing a string.

Returns a copy of the string from the event field with the name specified by *s*, but with all uppercase characters converted to lowercase. Any non-alphabetic or lowercase characters in *s* are returned unaltered.

Note:

This function supports locale sensitivity to the extent possible on your particular platform.

toString

```
string toString(  
    <type> field)
```

field The name of an event field to be converted.

Returns a string containing the event field *field*, converted to a string. Any supported field type, except for structures and sequences, may be converted.

Note:

The locale-specific radix character will be used when converting a floating point type.

toUpperCase

```
char toUpperCase(
    string s,
    int index)
```

s The name of an event field containing a string.

index The index of the character within the string field that is to be returned. This index is zero-based, so the first character has an index of 0.

Returns a copy of the string from the event field with the name specified by *s*, but with all lowercase characters converted to uppercase. Any non-alphabetic or uppercase characters in *s* are returned unaltered.

Note:

This function supports locale sensitivity to the extent possible on your particular platform.

toUpperCase

```
char toUpperCase(
    string s)
```

s The name of an event field containing a string.

Returns a single character from the event field with the name specified by *s*, converted to uppercase. If the specified character is non-alphabetic or is already uppercase, it is returned unaltered.

Note:

This function supports locale sensitivity to the extent possible on your particular platform.

Using Filters with Subscriptions

When you subscribe to a particular event type using the `awNewSubscription` function, you may specify an optional event filter string. The filter string you specify will be used by the Broker when determining if an event should be placed in your Broker client's event queue.

Before an event is placed in your Broker client's event queue, it must meet these criteria:

1. The event type must match the subscription's event type.
2. The contents of the event must match the filter specification

The following example shows how to create an event filter string and use it with an event subscription:

```
BrokerClient c;
char *filter_string;
. . .
```

```

filter_string = "(A<B) and ((C+12) > (D*3))";
. . .
/* Make a subscription */
err = awNewSubscription(c,"Sample::SimpleEvent",filter_string);
if (err != AW_NO_ERROR) {
. . .
}
. . .

```

Using BrokerFilters

Your client application may create a `BrokerFilter` using the `awNewBrokerFilter` function. A `BrokerFilter` may be used locally by the client application, in conjunction with the `awMatchFilter` function, to determine if a particular event type matches the filter.

When creating a `BrokerFilter`, you must specify an event type name and a filter string. You may then use the `awMatchFilter` function to check any events that your client application might receive.

The following example shows a code excerpt that creates three filters for use by a client application. The `awMatchFilter` function is then used to determine if the event matches each filter's criteria.

```

. . .
awNewBrokerFilter(my_client,"Type1",
    "(A<B) && ((C+12) >(D*3))",&f1);
awNewBrokerFilter(my_client, "Type1",
    NULL,&f2);
awNewBrokerFilter(my_client,"Type2",NULL,&f3);

awGetEvent(my_client,&my_event);
awMatchFilter(f1,my_event,&is_special);
awMatchFilter(f2,my_event,&is_of_type1);
awMatchFilter(f3,my_event,&is_of_type2);
if (is_special) {
    /* A special case of Type1 events that requires special processing */
    . . .
} else if (is_of_type1) {
    /* A Type1 event */
    . . .
} else if (is_of_type2) {
    /* A Type2 event */
    . . .
}

```

To do this same processing without `BrokerFilters`, your code would look like that shown in the following example:

```

. . .
awGetEvent(my_client,&my_event);
awGetEventTypeName(my_event,&name);
awGetIntegerField(my_event,"A",&a);
awGetIntegerField(my_event,"B",&b);
awGetIntegerField(my_event,"C",&c);
awGetIntegerField(my_event,"D",&d);
is_special = (strcmp(name,"Type1")==0) && (a<b) && ((c+12) >(d*3));
is_of_type1 = (strcmp(name,"Type1")==0);
is_of_type2 = (strcmp(name,"Type2")==0);
free(name);

```

...

Destroying Broker Filters

The [awDeleteFilter](#) method allows you to destroy a Broker filter.

Obtaining Filter Strings

You can use the [awGetFilterString](#) method to obtain the filter string associated with a Broker filter.

Obtaining Event Type Names

You can use the [awGetFilterEventTypeName](#) method to obtain the event type name associated with a Broker filter.

Converting Broker Filters to Strings

The [awGet<type>Field](#) method allows you to obtain a character string containing the contents of a Broker filter.

14 API Reference

■	awAbort	148
■	awAcknowledge	148
■	awBegin	150
■	awBroker	151
■	awCan	153
■	awCancel	154
■	awClear	161
■	awClient	163
■	awCommit	164
■	awCompare	164
■	awCopy	165
■	awDate	166
■	awDelete	167
■	awDeliver	169
■	awDescriptor	182
■	awDestroy	183
■	awDisconnect	184
■	awDispatch	186
■	awDoes	187

■	awEnd	188
■	awError	189
■	awEvent	190
■	awFilter	197
■	awFree	197
■	awFlush	198
■	awGet	198
■	awInit	279
■	awInterrupt	279
■	awIs	280
■	awLock	284
■	awMain	284
■	awMake	285
■	awMalloc	289
■	awMatch	290
■	awNative	290
■	awNew	291
■	awParse	311
■	awPrime	313
■	awPublish	314
■	awRealloc	319
■	awReconnect	319
■	awRegister	322
■	awResend	327

■	awSet	327
■	awSSL	362
■	awStop	362
■	awString	363
■	awThreaded	371
■	awTx	372
■	awType	394
■	awUnlock	394
■	awValidate	394

awAbort

awAbortTransactions

```
BrokerError awAbortTransactions(
    BrokerTxClient txclient,
    int n,
    BrokerTxClient *txclients);
```

<i>txclient</i>	The transactional Broker client that is aborting the transaction.
<i>n</i>	List of the transactional clients.
<i>txclients</i>	Aborts the transaction.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The client is not valid.

In addition, any of the communications errors can occur. If an error occurs, the local client object is still deleted from local memory.

awAcknowledge

awAcknowledge

```
BrokerError awAcknowledge(
    BrokerClient client,
    BrokerLong seqn);
```

<i>client</i>	The Broker client that is acknowledging the receipt of an event.
<i>seqn</i>	The sequence number of the event to acknowledge. A value of zero acknowledges all the events received from the Broker that were previously unacknowledged.

Acknowledges the receipt of a single event, specified by *seqn*, for the specified Broker client. If *seqn* is set to zero, all previously unacknowledged events are acknowledged. By acknowledging one or more events, a client ensures that the Broker will not send those events to the Broker client again.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_ACKNOWLEDGENT	The <i>seqn</i> is not a valid event to acknowledge.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.

Possible BrokerError major codes	Meaning
AW_ERROR_OUT_OF_RANGE	The parameter <i>seqn</i> is less than zero.

See also:

“awAcknowledgeThrough” on page 149

“awGetEventReceiptSequenceNumber” on page 229

awAcknowledgeEvents

```
BrokerError awAcknowledgeEvents(
    BrokerClient client,
    int n_s,
    BrokerLong *seqn);
```

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_ACKNOWLEDGENT	The <i>seqn</i> is not a valid event to acknowledge.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_OUT_OF_RANGE	The parameter <i>seqn</i> is less than zero.

In addition, any of the communications errors can occur.

awAcknowledgeThrough

```
BrokerError awAcknowledgeThrough(
    BrokerClient client,
    BrokerLong seqn);
```

client The Broker client that is acknowledging the receipt of an event.

seqn The sequence number of the event to acknowledge. A value of 0 acknowledges all the events received that were previously unacknowledged.

Acknowledges the receipt of all events received by a Broker client, up to and including the event with the sequence number specified by *seqn*. If *seqn* is set to 0, all previously unacknowledged events are acknowledged. By acknowledging the events it has received, a Broker client ensures that the Broker will not send those events to the Broker client again. To acknowledge a single event, use the `awAcknowledge` function.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_ACKNOWLEDGENT	The <i>seqn</i> is not a valid event to acknowledge.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_OUT_OF_RANGE	The <i>seqn</i> is less than zero.

See also:

“awAcknowledge” on page 148

“awGetEventReceiptSequenceNumber” on page 229

awBegin

awBeginTransaction

```
BrokerError awBeginTransaction(
    BrokerClient client,
    char *transaction_id,
    int required_level,
    int num_participants,
    char **participants,
    int *reply_tag);
```

<i>client</i>	The Broker client that is beginning the transaction.
<i>transaction_id</i>	The identifier of the transaction, created with the awMakeTransactionId function.
<i>required_level</i>	Indicates the requested level of the transaction being started, specified as one of the values in the table below.
<i>num_participants</i>	The number of client identifiers contained in the <i>participants</i> array. May be set to zero.
<i>participants</i>	An array of client identifiers that indicates which clients are allowed to interact with events published as part of the transaction.
<i>reply_tag</i>	If not NULL, indicates that an acknowledgement reply is requested and the tag value of the published event is returned. This parameter is used for output.

Publishes an `Adapter::beginTransaction` event for the specified transaction identifier, which begins a transaction. All subsequent events published by this client with the same transaction identifier will be considered part of a transaction.

If *reply_tag* is not NULL, it will be set with the tag value of the event published so that you can match it to the acknowledgment that will follow. If *reply_tag* is NULL, no acknowledgment reply will be sent.

After invoking this method, your application can send any number of additional events with the `transactionId` envelope field set to the specified transaction identifier.

The `required_level` parameter indicates the level of the transaction that your Broker client is requesting. If the adapter does not support the requested transaction level, an error will be returned. The `required_level` must be set to one of the following values:

Transaction Level	Meaning
<code>AW_TRANSACTION_LEVEL_ANY</code>	Requests any of the following levels of transaction support from an adapter.
<code>AW_TRANSACTION_LEVEL_PSEUDO</code>	Requests pseudo transaction support, described in “Transactional Client Processing with Adapters” on page 423.
<code>AW_TRANSACTION_LEVEL_BASIC</code>	Requests basic transaction support, described in “Transactional Client Processing with Adapters” on page 423.
<code>AW_TRANSACTION_LEVEL_CONVERSATIONAL</code>	Requests conversational transaction support, described in “Transactional Client Processing with Adapters” on page 423.

Finally, you should use the `awEndTransaction` function to close the transaction. See [“Transactional Client Processing with Adapters”](#) on page 423 for more information on transaction processing.

Possible BrokerError major codes	Meaning
<code>AW_ERROR_INVALID_CLIENT</code>	The <code>client</code> parameter is not NULL and has been destroyed or disconnected.
<code>AW_ERROR_NO_PERMISSION</code>	The <code>client</code> does not have permission to publish the <code>Adapter::beginTransaction</code> event type.
<code>AW_ERROR_NULL_PARAM</code>	The <code>transaction_id</code> parameter is NULL.

See also:

[“awEndTransaction”](#) on page 188

[“awMakeTransactionId”](#) on page 286

awBroker

awBrokerLongFromString

```
BrokerLong awBrokerLongFromString(
    char *str);
```

str The string that is to be converted to a BrokerLong.

Returns a BrokerLong containing the number represented by *str*.

See also:

“awBrokerLongToString” on page 152

awBrokerLongToString

```
char *awBrokerLongToString(  
    BrokerLong bl,  
    char *string);
```

bl The BrokerLong to be converted to a string.

string The string representation that is returned. This parameter is used for output.

Obtains a string representation of the specified BrokerLong. The buffer pointed to by *string* must have at least 24 bytes of space.

See also:

“awBrokerLongToString” on page 152

“awBrokerToNativeLong” on page 152

awBrokerToNativeLong

```
NativeLong awBrokerToNativeLong(  
    BrokerLong bl);
```

bl The BrokerLong to be converted to a native, 64-bit long.

Returns a NativeLong representing the number contained in *bl*.

Note:

This function is available only on platforms which support 64-bit integral representations.

See also:

“awGetBrokerLong” on page 203

“awNativeToBrokerLong” on page 290

“awSetBrokerLong” on page 331

awCan

awCanPublish

```
BrokerError awCanPublish(
    BrokerClient client,
    char *event_type_name,
    BrokerBoolean *can_publish);
```

<i>client</i>	The Broker client whose ability to publish is to be tested.
<i>event_type_name</i>	The event type name that the Broker client wishes to publish.
<i>can_publish</i>	A indication of whether or not the Broker client is permitted to publish the specified event type. Set to 1 (true) if the Broker client is permitted; otherwise, set to 0 (false). This parameter is used for output.

Determines whether or not the specified Broker client can publish or deliver the specified event type. Upon return, the *can_publish* parameter is set to 1 (true) or 0 (false). If set to 1 (true), the Broker client can publish or deliver events of this event type. Conversely, if set to 0 (false), the Broker client cannot publish or deliver such events.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>event_type_name</i> or <i>can_publish</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

- “awCanSubscribe” on page 153
- “awPublishEvent” on page 314
- “awPublishEvents” on page 315
- “awPublishEventsWithAck” on page 316

awCanSubscribe

```
BrokerError awCanSubscribe(
    BrokerClient client,
    char *event_type_name,
    BrokerBoolean *can_subscribe);
```

<i>client</i>	The Broker client whose ability to subscribe is to be tested.
---------------	---------------------------------------------------------------

<i>event_type_name</i>	The event type name to which the Broker client wishes to subscribe.
<i>can_subscribe</i>	An indication of whether or not the Broker client is permitted to subscribe to the specified event type. Set to 1 (true) if the Broker client is permitted; otherwise, set to 0 (false). This parameter is used for output.

Determines whether or not the specified *client* can subscribe to the specified event type. Also determines if an event of the specified type can be delivered to the *client*. Upon return, the *can_subscribe* parameter is set to 1 (true) if the Broker client is allowed to subscribe to the event; otherwise *can_subscribe* is set to 0 (false).

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>event_type_name</i> or <i>can_subscribe</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awCanPublish” on page 153

“awNewSubscription” on page 301

“awNewSubscriptionFromStruct” on page 302

“awNewSubscriptionsFromStructs” on page 303

“awNewSubscriptionWithId” on page 305

awCancel

awCancelCallbackForSubId

```
BrokerError awCancelCallbackForSubId(
    BrokerClient client,
    int sub_id);
```

<i>client</i>	The Broker client whose callback operation is to be cancelled.
<i>sub_id</i>	The subscription identifier whose callback function is to be cancelled.

Cancels the callback function for the specified subscription ID for this Broker client.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.

See also:

“awCancelCallbacks” on page 155

“awRegisterCallback” on page 322

“awRegisterCallbackForSubId” on page 323

awCancelCallbackForTag

```
BrokerError awCancelCallbackForTag(
    BrokerClient client,
    int tag);
```

client The Broker client for which the callback is to be cancelled.

tag The tag associated with the callback to be cancelled.

Cancels a callback function for the specified *tag* and *client*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.

See also:

“awCancelCallbacks” on page 155

“awRegisterCallback” on page 322

“awRegisterCallbackForTag” on page 324

awCancelCallbacks

```
BrokerError awCancelCallbacks(
    BrokerClient client);
```

Cancels all callback functions currently registered for the specified *client*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.

See also:

“awCancelCallbackForSubId” on page 154

“awCancelCallbackForTag” on page 155

“awRegisterCallback” on page 322

“awRegisterCallbackForSubId” on page 323

awCancelSubscription

```
BrokerError awCancelSubscription(
    BrokerClient client,
    char *event_type_name,
    char *filter);
```

<i>client</i>	The Broker client whose event subscription is to be cancelled.
<i>event_type_name</i>	The event type name of the subscription the Broker client wishes to cancel.
<i>filter</i>	The filter string that was used in the subscription. Set this to NULL if no filter was specified in the original subscription.

Cancels an event subscription with the specified event type and matching filter string for the specified Broker client.

You may add a wildcard character "*" to the end of the *event_type_name* to cancel multiple subscriptions within a particular event type scope. See [“Using Wildcards” on page 62](#) for more information.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	No matching subscription was found.
AW_ERROR_NULL_PARAM	The <i>event_type_name</i> is NULL.

See also:

“awCancelSubscriptionFromStruct” on page 157

“awCancelSubscriptionsFromStructs” on page 157

“awGetSubscriptions” on page 255

“awNewSubscription” on page 301

“awNewSubscriptionFromStruct” on page 302

“awNewSubscriptionsFromStructs” on page 303

“awNewSubscriptionWithId” on page 305

awCancelSubscriptionFromStruct

```
BrokerError awCancelSubscriptionFromStruct(
    BrokerClient client,
    BrokerSubscription *sub);
```

client The Broker client whose event subscription is to be cancelled.

sub The structure representing the subscription the Broker client wishes to cancel. See [“BrokerSubscription Objects” on page 67](#) for more information.

Cancels a subscription, specified by the subscription structure, for the specified Broker client. Only the `event_type_name` and `filter` fields of the subscription structure are used to find a matching subscription to cancel; the `subId` field is ignored.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	No matching subscription was found.
AW_ERROR_NULL_PARAM	The <i>sub</i> parameter is NULL or the <code>event_type_name</code> it contains is NULL.

See also:

“awCancelSubscription” on page 156

“awCancelSubscriptionsFromStructs” on page 157

“awGetSubscriptions” on page 255

“awNewSubscription” on page 301

“awNewSubscriptionFromStruct” on page 302

“awNewSubscriptionsFromStructs” on page 303

“awNewSubscriptionWithId” on page 305

awCancelSubscriptionsFromStructs

```
BrokerError awCancelSubscriptionsFromStructs(
    BrokerClient client,
    int n,
    BrokerSubscription *subs);
```

client The Broker client whose event subscription is to be cancelled.

n The number of subscription structures in the array.

subs Any array of subscription structures representing the subscriptions the Broker client wishes to cancel.

Cancels the *n* subscriptions, specified by the array of subscription structures *subs*, for the specified *client*. Only the *event_type_name* and *filter* fields of the subscription structure are used to find a matching subscription to cancel; the *subId* field is ignored.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	No matching subscription was found for one or more of <i>subs</i> .
AW_ERROR_NULL_PARAM	The parameter <i>subs</i> is NULL or an <i>event_type_name</i> field in one of the subscription structures is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>n</i> parameter is less than zero.

See also:

“awCancelSubscription” on page 156

“awCancelSubscriptionFromStruct” on page 157

“awGetSubscriptions” on page 255

“awNewSubscription” on page 301

“awNewSubscriptionFromStruct” on page 302

“awNewSubscriptionsFromStructs” on page 303

“awNewSubscriptionWithId” on page 305

awCancelTxSubscription

```
BrokerError awCancelTxSubscription(
    BrokerTxClient txclient,
    char *event_type_name,
    char *filter);
```

txclient The Broker client whose event subscription is to be cancelled.

event_type_name The event type name of the subscription the Broker client wishes to cancel.

filter The filter string that was used in the subscription. Set this to NULL if no filter was specified in the original subscription.

The *txclient* cancels an event subscription with the specified event type and matching filter string for the specified Broker client.

You may add a wildcard character "*" to the end of the *event_type_name* to cancel multiple subscriptions within a particular event type scope. See [“Using Wildcards” on page 62](#) for more information.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	No matching subscription was found.
AW_ERROR_NULL_PARAM	The <i>event_type_name</i> is NULL.

See also:

“awCancelTxSubscriptionFromStruct” on page 159

“awCancelTxSubscriptionsFromStructs” on page 160

“awGetSubscriptions” on page 255

“awNewTxSubscription” on page 306

“awNewTxSubscriptionFromStruct” on page 307

“awNewTxSubscriptionsFromStructs” on page 308

“awNewTxSubscriptionWithId” on page 309

awCancelTxSubscriptionFromStruct

```
BrokerError awCancelTxSubscriptionFromStruct(
    BrokerTxClient txclient,
    BrokerSubscription *sub);
```

txclient The Broker client whose event subscription is to be cancelled.

sub The structure representing the subscription the Broker client wishes to cancel. See [“BrokerSubscription Objects” on page 67](#) for more information.

The transactional client cancels a subscription, specified by the subscription structure, for the specified Broker client. Only the *event_type_name* and *filter* fields of the subscription structure are used to find a matching subscription to cancel; the *subId* field is ignored.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	No matching subscription was found.
AW_ERROR_NULL_PARAM	The <i>sub</i> parameter is NULL or the <i>event_type_name</i> it contains is NULL.

See also:

- “awCancelTxSubscription” on page 158
- “awCancelTxSubscriptionsFromStructs” on page 160
- “awGetSubscriptions” on page 255
- “awNewTxSubscription” on page 306
- “awNewTxSubscriptionFromStruct” on page 307
- “awNewTxSubscriptionsFromStructs” on page 308
- “awNewTxSubscriptionWithId” on page 309

awCancelTxSubscriptionsFromStructs

```
BrokerError awCancelTxSubscriptionsFromStructs(
    BrokerTxClient txclient,
    int n,
    BrokerSubscription *subs);
```

<i>txclient</i>	The Broker client whose event subscription is to be cancelled.
<i>n</i>	The number of subscription structures in the array.
<i>subs</i>	Any array of subscription structures representing the subscriptions the Broker client wishes to cancel.

Cancels the *n* subscriptions, specified by the array of subscription structures *subs*, for the specified *txclient*. Only the *event_type_name* and *filter* fields of the subscription structure are used to find a matching subscription to cancel; the *subId* field is ignored.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	No matching subscription was found for one or more of <i>subs</i> .
AW_ERROR_NULL_PARAM	The parameter <i>subs</i> is NULL or an <i>event_type_name</i> field in one of the subscription structures is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>n</i> parameter is less than zero.

See also:

- “awCancelSubscription” on page 156
- “awCancelSubscriptionFromStruct” on page 157

“awGetSubscriptions” on page 255

“awNewSubscription” on page 301

“awNewSubscriptionFromStruct” on page 302

“awNewSubscriptionsFromStructs” on page 303

“awNewSubscriptionWithId” on page 305

awClear

awClearClientQueue

```
BrokerError awClearClientQueue(
    BrokerClient client);
```

Clears all events in the queue for the specified *client*.

Important:

Use this function with care because it deletes events that have not yet been processed.

Possible BrokerError major codes

Meaning

AW_ERROR_INVALID_CLIENT

The *client* has been destroyed or disconnected.

See also:

“awGetClientQueueLength” on page 213

awClearDate

```
void awClearDate(
    BrokerDate *d);
```

Clears the date and time setting in the specified *BrokerDate*, *d*.

See also:

“awClearTime” on page 163

“awSetDate” on page 334

“awSetTime” on page 354

“awSetDateTime” on page 335

awClearEvent

```
BrokerError awClearEvent(  
    BrokerEvent event);
```

Clears all fields of the specified *event*, so it may be reused.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.

See also:

“awClearEventField” on page 162

awClearEventField

```
BrokerError awClearEventField(  
    BrokerEvent event,  
    char *field_name);
```

event The event whose field is to be cleared.

field_name The name of the field to be cleared.

Clears the field *field_name* in the specified *event*. Note that *field_name* may directly identify any field in an event no matter how deeply nested the events may be.

See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> is NULL.

See also:

“awClearEvent” on page 162

“awGetFieldNames” on page 242

awClearTime

```
void awClearTime(
    BrokerDate *d);
```

Clears the time information in the specified date, but leaves the date information intact.

See also:

“awClearDate” on page 161

“awSetDate” on page 334

“awSetTime” on page 354

“awSetDateTime” on page 335

awClearTxClientQueue

```
BrokerError awClearTxClientQueue(
    BrokerTxClient txclient);
```

Clears all events in the queue for the specified *txclient*.

Important:

Use this function with care because it deletes events that have not yet been processed.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.

See also:

“awGetTxClientQueueLength” on page 262

awClient

awClientLastPublishSequenceNumber

```
BrokerError awGetClientLastPublishSequenceNumber(
    BrokerClient client,
    BrokerLong *seqn);
```

<i>client</i>	The Broker client whose last published sequence number is to be returned.
<i>seqn</i>	The sequence number last published by the Broker client. This parameter is used for output.

Obtains the highest sequence number used by the *client* for publishing or delivering an event.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>seqn</i> parameter is NULL.

awClientToString

```
char * awClientToString(
    BrokerClient client);
```

Returns a string that contains names of the Broker client, client group, and Broker of the specified *client*. The string is suitable for display on a computer screen. The caller is responsible for freeing the returned value.

awCommit

awCommitTransactions

```
BrokerError awCommitTransactions(
    BrokerTxClient txclient,
    int n,
    BrokerTxClient *txclients);
```

Commits the list of transactions.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.

In addition, any of the communications errors can occur, but if they do, the local client object is still deleted from local memory.

awCompare

awCompareDate

```
int awCompareDate(
    BrokerDate *date1,
    BrokerDate *date2);
```

date1 One date to be compared.

date2 The other date.

Compares the date and time represented by *date1* and *date2*. Returns:

- Zero if *date1* is equal to *date2*
- -1 if *date1* is earlier than *date2*
- 1 if *date1* is later *date2*

See also:

“awDeleteDate” on page 167

“awNewBrokerDate” on page 293

awCopy

awCopyDate

```
BrokerDate * awCopyDate(
    BrokerDate *d);
```

Returns a new date, initialized with the information in the specified date *d*. The caller is responsible for calling `awDeleteDate` to free the returned `BrokerDate`.

See also:

“awDeleteDate” on page 167

“awNewBrokerDate” on page 293

“awNewEmptyBrokerDate” on page 297

awCopyDescriptor

```
BrokerConnectionDescriptor awCopyDescriptor(
    BrokerConnectionDescriptor desc);
```

desc The descriptor to copy.

Returns a copy of the connection descriptor, *desc*. The caller is responsible for calling `awDeleteDescriptor` on the return value.

This function returns `NULL` if memory cannot be allocated or if *desc* is invalid.

See also:

“awDeleteDescriptor” on page 168

“awSetClientStateShareLimit” on page 333

awCopyError

```
BrokerError awCopyError(  
    BrokerError err);
```

err The error to copy.

Creates a copy of the specified error *err*. The caller is responsible for calling `awDeleteError` on the return value.

This function returns `NULL` if memory cannot be allocated or if *err* is invalid.

See also:

“awDeleteError” on page 168

“awSetClientStateShareLimit” on page 333

awCopyEvent

```
BrokerEvent awCopyEvent(  
    BrokerEvent event);
```

event The event that is to be copied.

Returns a copy of the specified *event*. The caller is responsible for calling `awDeleteEvent` on the returned value.

This function returns `NULL` if the *event* to be copied was invalid or if the event parameter was `NULL`.

See also:

“awDeleteEvent” on page 168

“awNewBrokerEvent” on page 294

awDate

awDateToLocalizedString

```
char * awDateToLocalizedString(  
    BrokerDate *d);
```

Converts the specified date *d* to string format, using the current locale, and returns the string. The caller is responsible for calling `free` on the returned value.

Note:

Milliseconds may not appear in the resulting string, depending on the current locale.

See also:

“`awDateToString`” on page 167

awDateToString

```
char * awDateToString(
    BrokerDate *d);
```

Converts the specified date *d* to string format and returns the string. The caller is responsible for calling `free` on the returned value.

Note:

Milliseconds will not appear in the resulting string if they are zero.

See also:

“`awDateToLocalizedString`” on page 166

awDelete

awDeleteClientQueueLock

```
void awDeleteClientQueueLock(ClientQueueLock qlock);
```

Deletes the `qlock` object by freeing allocated members. It is the responsibility of the user to release the lock before deleting the lock.

Returns `AW_ERROR_INVALID_QUEUE_LOCK` when an invalid `qlock` object is passed. Otherwise returns appropriate information.

awDeleteDate

```
void awDeleteDate(
    BrokerDate *d);
```

Deletes the specified date *d* and releases its storage.

See also:

“`awCopyDate`” on page 165

“awNewBrokerDate” on page 293

awDeleteDescriptor

```
void awDeleteDescriptor(  
    BrokerConnectionDescriptor desc);
```

Deletes the specified connection descriptor, *desc*.

See also:

“awCopyDescriptor” on page 165

“awNewBrokerConnectionDescriptor” on page 292

awDeleteError

```
void awDeleteError(  
    BrokerError err);
```

err The error to be deleted.

Deletes the specified error *err*.

Note:

Only use this function to delete an error object created by the `awCopyError` function.

See also:

“awCopyError” on page 166

awDeleteEvent

```
void awDeleteEvent(  
    BrokerEvent event);
```

Deletes the specified *event* and releases its storage.

Important:

Do not invoke this function from within a callback function to delete a received event. Other callback functions may be invoked later to process the same event.

See also:

“awCopyEvent” on page 166

“awNewBrokerEvent” on page 294

awDeleteFilter

```
void awDeleteFilter(  
    BrokerFilter filter);
```

Deletes the specified *filter* and releases its storage.

See also:

“awNewBrokerFilter” on page 295

awDeleteString

```
void awDeleteString(  
    BrokerString st);
```

Deletes the specified string *st*.

See also:

“awDeleteStringWrapper” on page 169

“awNewBrokerString” on page 296

awDeleteStringWrapper

```
void awDeleteStringWrapper(  
    BrokerString st);
```

Deletes the string wrapper for the specified string *st*, but does not delete the string it contains. This function is useful if you extract a string with `awStringPointer` and you no longer need its wrapper.

See also:

“awDeleteString” on page 169

“awNewBrokerString” on page 296

“awStringPointer” on page 371

awDeliver

awDeliverAckReplyEvent

```
BrokerError awDeliverAckReplyEvent(  
    BrokerClient client,  
    BrokerEvent request_event,  
    BrokerLong publish_seqn);
```

<i>client</i>	The Broker client that is delivering the event.
<i>request_event</i>	The original request event. Used to determine the client ID of the sender of the original request.
<i>publish_seqn</i>	The publish sequence number to use on the reply event. Should be set to zero if you are not using publish sequence numbers.

Delivers an `Adapter::ack` event to the originator of the specified *request_event*. This function properly sets the `tag` envelope field to match that of the request.

If the `trackId` envelope field was set on *request_event*, that value is copied to the `Adapter::ack` event. If the *request_event* `trackId` envelope field was not set, the `pubId` envelope field from the *request_event* will be used instead.

The Broker client that is to receive the delivered event is not required to have registered a subscription for the event type, but its client group must allow the Broker client to receive the event type.

Note:

An error will not be returned if the recipient, represented by the `pubId` field in *request_event*, no longer exists.

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_INVALID_CLIENT</code>	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
<code>AW_ERROR_INVALID_EVENT</code>	The <i>request_event</i> is invalid.
<code>AW_ERROR_NO_PERMISSION</code>	The <i>client</i> does not have permission to publish <code>Adapter::ack</code> .

See also:

“`awDeliverErrorReplyEvent`” on page 170

“`awDeliverEvents`” on page 173

“`awDeliverNullReplyEvent`” on page 176

“`awDeliverPartialReplyEvents`” on page 177

“`awDeliverReplyEvent`” on page 179

“`awDeliverReplyEvents`” on page 180

awDeliverErrorReplyEvent

```
BrokerError awDeliverErrorReplyEvent(
    BrokerClient client,
```

```
BrokerEvent request_event,
BrokerEvent error_event);
```

<i>client</i>	The Broker client that is delivering the event.
<i>request_event</i>	The original request event. Used to determine the client ID of the sender of the original request.
<i>error_event</i>	The error event to be delivered.

Sends a single error event to the Broker to be delivered to the Broker client that published the *request_event*. This function properly sets the `tag`, `appSeqn`, and `appLastSeqn` envelope fields.

If the `trackId` envelope field was set on *request_event*, that value is copied to the *error_event* event. If the *request_event* `trackId` envelope field was not set, the `pubId` envelope field from the *request_event* will be used instead.

The *error_event* will be delivered to the Broker client with the client ID contained in the `errorsTo` envelope field of *request_event*, if it was set by the requestor.

Note:

An error will not be returned if the recipient, represented by the `pubId` field in *request_event*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <i>request_event</i> or <i>error_event</i> is invalid, the <i>request_event</i> was not received from the Broker, or the <i>error_event</i> does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish the <i>error_event</i> .
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for <i>error_event</i> does not exist on the Broker.

See also:

“awDeliverAckReplyEvent” on page 169

“awDeliverEvents” on page 173

“awDeliverNullReplyEvent” on page 176

“awDeliverPartialReplyEvents” on page 177

“awDeliverReplyEvent” on page 179

“awDeliverReplyEvents” on page 180

awDeliverEvent

```
BrokerError awDeliverEvent(
    BrokerClient client,
    char *dest_id,
    BrokerEvent event);
```

client The Broker client that is delivering the event.

dest_id Identifies the Broker client to which the event is to be delivered.

event The event that is to be delivered.

Sends the specified *event* to the Broker client with the client identifier represented by *dest_id*. The event is sent to the Broker which, in turn, forwards it to the destination Broker client. The *client* that is to receive the delivered event is not required to have registered a subscription for the event type, but its client group must allow the Broker client to receive the event type.

A typical use of this function is when your Broker client replies to a request event from another Broker client. In such a case, you can obtain the *dest_id* by extracting it from the envelope of the request event, as described in [“Delivering Events” on page 72](#).

Note:

An error will not be returned if the recipient, represented by *dest_id*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_CLIENT_ID	The destination client ID contains illegal characters
AW_ERROR_INVALID_EVENT	The <i>event</i> is not valid.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish the event type.
AW_ERROR_NULL_PARAM	The parameter <i>dest_id</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for the event does not exist on the Broker.

See also:

“awDeliverAckReplyEvent” on page 169

“awDeliverEvents” on page 173

“awDeliverNullReplyEvent” on page 176

“awDeliverPartialReplyEvents” on page 177

“awDeliverReplyEvents” on page 180

“awPublishEvent” on page 314

“awPublishEvents” on page 315

awDeliverEvents

```
BrokerError awDeliverEvents(
    BrokerClient client,
    char *dest_id,
    int n,
    BrokerEvent *events);
```

<i>client</i>	The Broker client that is sending the events.
<i>dest_id</i>	The identifier of the Broker client to which the events are to be delivered.
<i>n</i>	The number of events in the array.
<i>events</i>	The array of events that are to be delivered.

Deliver multiple events. Gives an array of events to the Broker to have them all delivered to the client with the given client ID. Either all of the events or none of them are delivered. 'n' is the number of events in the events array. No error is returned if there is no client using the destination client ID.

Note:

An error will not be returned if the recipient, represented by *dest_id*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_CLIENT_ID	The destination client ID contains illegal characters
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish the event type.
AW_ERROR_NULL_PARAM	The parameter <i>dest_id</i> or <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awDeliverAckReplyEvent” on page 169

“awDeliverErrorReplyEvent” on page 170

“awDeliverEvents” on page 173

“awDeliverEventsWithAck” on page 174

“awDeliverNullReplyEvent” on page 176

“awDeliverPartialReplyEvents” on page 177

“awDeliverReplyEvent” on page 179

“awDeliverReplyEvents” on page 180

“awPublishEvent” on page 314

“awPublishEvents” on page 315

“awPublishEventsWithAck” on page 316

awDeliverEventsWithAck

```
BrokerError awDeliverEventsWithAck(  
    BrokerClient client,  
    char *dest_id,  
    int n,  
    BrokerEvent *events,  
    int ack_type,  
    int n_acks,  
    BrokerLong *ack_seqn);
```

<i>client</i>	The Broker client that wishes to publish the events.
<i>dest_id</i>	The identifier of the Broker client to which the events are to be delivered.
<i>n</i>	The number of events in the events array.
<i>events</i>	The array of events to be delivered.
<i>ack_type</i>	Determines how the events will be acknowledged and must be set to one of the following: <ul style="list-style-type: none">■ AW_ACK_NONE■ AW_ACK_AUTOMATIC■ AW_ACK_THROUGH■ AW_ACK_SELECTIVE
<i>n_acks</i>	The number of sequence numbers in <i>ack_seqn</i> .
<i>ack_seqn</i>	The array of sequence numbers to be acknowledged if <i>ack_type</i> is set to AW_ACK_THROUGH or AW_ACK_SELECTIVE.

Sends the array of *events* to the Broker for delivery to the Broker client destination specified by *dest_id*. You also have one of several options for acknowledging events already received by this client. Either all events or none will be delivered.

Note:

An error will not be returned if the recipient, represented by *dest_id*, no longer exists.

The setting of the *ack_type* and *ack_seqn* parameters will determine which events received by this client are to be acknowledged.

ack_type	ack_seqn	Result
AW_ACK_NONE	Not applicable.	No events are acknowledged.
AW_ACK_AUTOMATIC	Not applicable.	All events received by the client are acknowledged.
AW_ACK_THROUGH	<i>ack_seqn</i> [0] contains the sequence number of the last event to be acknowledged. If set to 0, the behavior will be the same as AW_ACK_AUTOMATIC	Acknowledges all events up to and including the sequence number specified by <i>ack_seqn</i> [0]. If the <i>n_acks</i> argument is zero, no events will be acknowledged.
AW_ACK_SELECTIVE	<i>ack_seqn</i> contains the sequence numbers of the specific events to be acknowledged.	Acknowledges the specific events whose sequence number are contained in <i>ack_seqn</i> . The <i>n_acks</i> argument must specify the number of sequence numbers contained in <i>ack_seqn</i> . All sequence numbers must be greater than zero.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_ACKNOWLEDGEMENT	The parameter <i>ack_seqn</i> contained an invalid sequence number. The events were not sent to the Broker.
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_INVALID_CLIENT_ID	The <i>dest_id</i> contains invalid characters.
AW_ERROR_INVALID_EVENT	One of the events contained in <i>events</i> is invalid.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish all of the event types.
AW_ERROR_NULL_PARAM	The parameter <i>events</i> or <i>dest_id</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awCanPublish” on page 153

“awDeliverErrorReplyEvent” on page 170

“awDeliverEvents” on page 173

“awPublishEvent” on page 314

“awPublishEvents” on page 315

“awPublishEventsWithAck” on page 316

awDeliverNullReplyEvent

```
BrokerError awDeliverNullReplyEvent(
    BrokerClient client,
    BrokerEvent request_event,
    char *reply_event_type_name,
    BrokerLong publish_seqn);
```

<i>client</i>	The Broker client that is delivering the reply event.
<i>request_event</i>	The original request event. Used to determine the client ID of the sender of the original request.
<i>reply_event_type_name</i>	The type name of the null reply event.
<i>publish_seqn</i>	The publish sequence number for the reply event. Set to zero if your application is not using publish sequence numbers.

Delivers a null event of type *reply_event_type_name* to the publisher of the *request_event*. The envelope tag, appSeqn, and appLastSeqn fields are set to indicate that this is a null event. This indicates that the request was successful and resulted in no data.

If the `trackId` envelope field was set on *request_event*, that value is copied to the null reply event. If the *request_event* `trackId` envelope field was not set, the `pubId` envelope field from the *request_event* will be used instead.

Note:

An error will not be returned if the recipient, represented by the `pubId` field in *request_event*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <i>request_event</i> is invalid.

Possible BrokerError major codes	Meaning
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish the event type.
AW_ERROR_NULL_PARAM	The parameter <i>reply_event_type_name</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for the reply event does not exist on the Broker.

See also:

“awDeliverAckReplyEvent” on page 169

“awDeliverErrorReplyEvent” on page 170

“awDeliverEvents” on page 173

“awDeliverPartialReplyEvents” on page 177

“awDeliverReplyEvent” on page 179

“awDeliverReplyEvents” on page 180

awDeliverPartialReplyEvents

```
BrokerError awDeliverPartialReplyEvents(
    BrokerClient client,
    BrokerEvent request_event,
    int n,
    BrokerEvent *events,
    int flag,
    int *reply_token);
```

<i>client</i>	The Broker client that is delivering the events.
<i>request_event</i>	The original request event. Used to determine the client ID of the sender of the original request.
<i>n</i>	The number of events in the reply.
<i>events</i>	The array of reply events that are to be delivered.
<i>flag</i>	Indicates if there are more events to be sent as part of this reply. Must be one of the following: <ul style="list-style-type: none"> ■ AW_REPLY_FLAG_START ■ AW_REPLY_FLAG_CONTINUE ■ AW_REPLY_FLAG_START_AND_END ■ AW_REPLY_FLAG_END

reply_token The address of a `int` that can be used by the function as temporary storage between calls.

Delivers an *event* array of size *n* to the Broker to be delivered to the Broker client that originally published *request_event*. No error will be returned if the Broker client using that ID no longer exists. Either all of the events or none of them will be delivered. This function properly sets the `tag`, `appSeqn`, and `appLastSeqn` envelope fields.

If the `trackId` envelope field was set on *request_event*, that value is copied to the reply events. If the *request_event* `trackId` envelope field was not set, the `pubId` envelope field from the *request_event* will be used instead.

This function is used to deliver parts of a set of replies in groups. When called the first time, *flag* should be `AW_REPLY_FLAG_START`. After doing this, additional calls can be made with other flag values. During intermediate replies, *flag* should be `AW_REPLY_FLAG_CONTINUE`. On the final call, *flag* should be `AW_REPLY_FLAG_END`. It is important that the ending call be made with *n* is set to at least 1.

Calling this function with *flag* set to `AW_REPLY_FLAG_START_AND_END` allows the entire result to be passed to this function in one call.

The *reply_token* value will be set and modified by this function during calls. It exists to carry information between calls and has no meaning to the caller.

Note:

An error will not be returned if the recipient, represented by the `pubId` field in *request_event*, no longer exists.

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_INVALID_CLIENT</code>	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
<code>AW_ERROR_INVALID_EVENT</code>	The <i>request_event</i> or one of the reply events is invalid, <i>request_event</i> was not received from the Broker, or one the reply events does not match its type definition.
<code>AW_ERROR_NO_PERMISSION</code>	The <i>client</i> does not have permission to publish the <i>reply_event</i> type.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>events</i> or <i>reply_token</i> is <code>NULL</code> .
<code>AW_ERROR_OUT_OF_RANGE</code>	The parameter <i>n</i> is less than zero or <i>flag</i> does not contain a valid value.
<code>AW_ERROR_UNKNOWN_EVENT_TYPE</code>	The event type for one of the reply events does not exist on the Broker.

See also:

“awDeliverAckReplyEvent” on page 169

“awDeliverErrorReplyEvent” on page 170

“awDeliverEvents” on page 173

“awDeliverNullReplyEvent” on page 176

“awDeliverReplyEvent” on page 179

“awDeliverReplyEvents” on page 180

awDeliverReplyEvent

```
BrokerError awDeliverReplyEvent(
    BrokerClient client,
    BrokerEvent request_event,
    BrokerEvent event);
```

<i>client</i>	The Broker client that is sending the event.
<i>request_event</i>	The original request event for which this reply is to be delivered.
<i>event</i>	The reply event to be delivered.

Transfers the *event* to the Broker to be delivered to the Broker client who sent the original *request_event*. No error is returned if the Broker client with the destination identifier specified in *request_event* no longer exists. This function properly sets the `tag`, `appSeqn`, and `appLastSeqn` envelope fields on *event*.

If the `trackId` envelope field was set on *request_event*, that value is copied to the reply event. If the *request_event* `trackId` envelope field was not set, the `pubId` envelope field from the *request_event* will be used instead.

Note:

An error will not be returned if the recipient, represented by the `pubId` field in *request_event*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <i>request_event</i> or the reply <i>event</i> is invalid, <i>request_event</i> was not received from the Broker, or the reply <i>event</i> does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish the <i>reply_event</i> type.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for <i>event</i> does not exist on the Broker.

See also:

“awDeliverErrorReplyEvent” on page 170

“awDeliverEvents” on page 173

“awDeliverAckReplyEvent” on page 169

“awDeliverNullReplyEvent” on page 176

“awDeliverReplyEvents” on page 180

awDeliverReplyEvents

```
BrokerError awDeliverReplyEvents(
    BrokerClient client,
    BrokerEvent request_event,
    int n,
    BrokerEvent *events);
```

<i>client</i>	The Broker client that is sending the event.
<i>request_event</i>	The original request event for which this reply is to be delivered.
<i>n</i>	The number of reply events to be delivered.
<i>events</i>	An array of reply event to be delivered.

Transfers the multiple *events* to the Broker to be delivered to the Broker client who sent the original *request_event*. This function properly sets the `tag`, `appSeqn`, and `appLastSeqn` envelope fields on all the events contained in the *events* array.

If the `trackId` envelope field was set on *request_event*, that value is copied to the reply events. If the *request_event* `trackId` envelope field was not set, the `pubId` envelope field from the *request_event* will be used instead.

Note:

An error will not be returned if the recipient, represented by the `pubId` field in *request_event*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <i>request_event</i> or one of the reply <i>events</i> is invalid, <i>request_event</i> was not received from the Broker, or one the reply <i>events</i> does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish the <i>reply_event</i> type.

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAM	The parameter <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for <i>event</i> does not exist on the Broker.

See also:

“awDeliverErrorReplyEvent” on page 170

“awDeliverEvents” on page 173

“awDeliverAckReplyEvent” on page 169

“awDeliverNullReplyEvent” on page 176

“awDeliverReplyEvent” on page 179

awDeliverRequestAndWait

```
BrokerError awDeliverRequestAndWait(
    BrokerClient client,
    char *dest_id,
    BrokerEvent event,
    int msec,
    int *n,
    BrokerEvent **reply_events);
```

<i>client</i>	The Broker client that is sending the event.
<i>dest_id</i>	The identifier of the Broker client to which the event is to be delivered.
<i>event</i>	The request event to be delivered.
<i>msec</i>	The number of milliseconds to block awaiting a reply. If set to AW_INFINITE, this function will block indefinitely.
<i>n</i>	The number of reply events returned in the <i>events</i> array. This parameter is used for output.
<i>reply_events</i>	The array of reply events that are returned. This parameter is used for output.

Sends the *event* to the Broker to be delivered to the Broker client specified by *dest_id* and then waits for all replies to be received. The last reply event is detected when an event is received with the envelope fields `appSeqn` and `appLastSeqn` being equal.

This function creates a value for the `tag` envelope field using the [awMakeTag](#) function. This function blocks until the replies are received or until the requested time-out interval expires.

You are responsible for deleting each event returned in *reply_events*, using the [awDeleteEvent](#) function, and for freeing the array itself, using *free*.

See “Using Request-Reply” on page 91 for more information on using the request-reply model.

Note:

You must register a general callback object, using the [awRegisterCallback](#) function, before calling this function. Also note that an error will not be returned if the recipient, represented by *dest_id*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	This function was called from a callback function.
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid or does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish the event type.
AW_ERROR_NULL_PARAM	The parameter <i>dest_id</i> , <i>n</i> , or <i>reply_events</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for <i>event</i> does not exist on the Broker.

See also:

“awDeliverReplyEvents” on page 180

“awPublishEvent” on page 314

“awPublishRequestAndWait” on page 318

awDescriptor

awDescriptorToString

```
char * awDescriptorToString(
    BrokerConnectionDescriptor desc);
```

desc The connection descriptor to be converted.

Returns the values of the connection descriptor *desc* in a string form suitable for display on a computer screen. The caller is responsible for freeing the memory associated with the return value.

Returns NULL if the *desc* is invalid or if memory could not be allocated.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The client is not valid.
AW_ERROR_INVALID_CLIENT_ID	The destination ID contains illegal characters.
AW_ERROR_INVALID_EVENT	The event is not valid, or if the event does not match its type definition.
AW_ERROR_NO_PERMISSION	The client does not have permission to publish the event type.
AW_ERROR_NULL_PARAM	The <code>dest_id</code> is NULL.
AW_ERROR_OUT_OF_RANGE	<code>n</code> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for the event does not exist on the Broker.

In addition, any of the communications errors can occur.

See also:

“`awNewBrokerConnectionDescriptor`” on page 292

“`awDeleteDescriptor`” on page 168

awDestroy

awDestroyClient

```
BrokerError awDestroyClient(
    BrokerClient client);
```

client The Broker client to be destroyed.

Disconnects the specified *client* and then destroys the Broker client, including the event queue allocated for the Broker client in the Broker.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.

See also:

“`awDisconnectClient`” on page 184

“`awNewBrokerClient`” on page 291

“`awReconnectBrokerClient`” on page 319

awDestroyTxClient

```
BrokerError awDestroyTxClient(
    BrokerTxClient txclient);
```

txclient The Broker client to be destroyed.

Disconnects the specified client and then destroys the transactional Broker client, including the event queue allocated for the Broker client in the Broker.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> is not valid.

See also:

“awDisconnectTxClient” on page 185

“awNewBrokerTxClient” on page 296

“awReconnectBrokerTxClient” on page 321

awDisconnect

awDisconnectClient

```
BrokerError awDisconnectClient(
    BrokerClient client);
```

client The Broker client to be disconnected.

Closes the connection between the specified *client* and the Broker and frees any local memory allocated for *client*. The effect of this function depends on the *life cycle* of the Broker client, which is defined by the *client group* to which the Broker client belongs. Client groups are described in *Administering webMethods Broker*.

- If the Broker client's life cycle is *destroy-on-disconnect*, the client state associated this Broker client is destroyed when it is disconnected. When the state is destroyed, any queued events and subscriptions are destroyed.
- If the Broker client's life cycle is *explicit-destroy*, the client's state on the Broker will persist until the Broker client is explicitly destroyed with the `awDestroyClient` function. In this case, calling `awDisconnectClient` simply breaks the Broker client's connection to the Broker. After the Broker client is disconnected:
 - The associated client state continues to exist.
 - The Broker continues to queue messages.

- The Broker client can resume processing events by reconnecting to the Broker.

Use the [awReconnectBrokerClient](#) function to reconnect a disconnected Broker client.

Note:

Your Broker client will be disconnected, even if an exception is thrown.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.

See also:

“awDisconnectClient” on page 184

“awNewBrokerClient” on page 291

“awReconnectBrokerClient” on page 319

awDisconnectTxClient

```
BrokerError awDisconnectTxClient(
    BrokerTxClient txclient);
```

txclient The transactional Broker client to be disconnected.

Closes the connection between the specified *txclient* and the Broker and frees any local memory allocated for *txclient*. The effect of this function depends on the *life cycle* of the Broker client, which is defined by the *client group* to which the Broker client belongs. Client groups are described in *Administering webMethods Broker*.

- If the Broker client's life cycle is *destroy-on-disconnect*, the client state associated this Broker client is destroyed when it is disconnected. When the state is destroyed, any queued events and subscriptions are destroyed.
- If the Broker client's life cycle is *explicit-destroy*, the client's state on the Broker will persist until the Broker client is explicitly destroyed with the [awDestroyClient](#) function. In this case, calling [awDisconnectClient](#) simply breaks the Broker client's connection to the Broker. After the Broker client is disconnected:
 - The associated client state continues to exist.
 - The Broker continues to queue messages.
 - The Broker client can resume processing events by reconnecting to the Broker.

Use the [awReconnectBrokerClient](#) function to reconnect a disconnected Broker client.

Note:

Your Broker client will be disconnected, even if an exception is thrown.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.

See also:

“awDisconnectTxClient” on page 185

“awNewBrokerConnectionDescriptor” on page 292

“awReconnectBrokerTxClient” on page 321

awDispatch

awDispatch

```
BrokerError awDispatch(int msec);
```

msec The number of milliseconds to wait for an event to process before timing out. If set to AW_INFINITE, this function will wait indefinitely.

Waits for an incoming event that needs to be passed to a callback function. This function will return after an event is received and passed to a callback function or after the wait time, specified by *msec*, has expired. If the wait time expires, this function returns an error with error code AW_ERROR_TIMEOUT.

When using `awDispatch`, you cannot invoke any of the following functions:

- [awMainLoop](#)
- [awThreadedCallbacks](#)

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	This function was called from a callback function.
AW_ERROR_INTERRUPTED	The awInterruptDispatch function was invoked.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_OUT_OF_RANGE	The <i>msec</i> parameter is less than -1.
AW_ERROR_TIMEOUT	The <i>msec</i> time-out interval expired before an event arrived.

See also:

“awGetEvent” on page 226

“awGetEvents” on page 230

“awGetEventsWithAck” on page 231

“awInterruptDispatch” on page 279

“awMainLoop” on page 284

“awThreadedCallbacks” on page 371

awDoes

awDoesSubscriptionExist

```
BrokerError awDoesSubscriptionExist(
    BrokerClient client,
    char *event_type_name,
    char *filter,
    BrokerBoolean *exists);
```

<i>client</i>	The Broker client.
<i>event_type_name</i>	The event type name for the subscription.
<i>filter</i>	The event filter string.
<i>exists</i>	On return this is set to 1 (true) if the subscription exists on the Broker, otherwise it is set to 0 (false). This parameter is used for output.

Determines if a subscription with the specified *event_type_name* and *filter* exists on the Broker for the *client*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>event_type_name</i> or <i>exists</i> is NULL.

See also:

“awCancelSubscription” on page 156

“awNewSubscription” on page 301

awDoesTxSubscriptionExist

```
BrokerError awDoesTxSubscriptionExist(
    BrokerTxClient txclient,
    char *event_type_name,
    char *filter,
    BrokerBoolean *exists);
```

<i>txclient</i>	The Broker client.
<i>event_type_name</i>	The event type name for the subscription.
<i>filter</i>	The event filter string.
<i>exists</i>	On return this is set to 1 (true) if the subscription exists on the Broker, otherwise it is set to 0 (false). This parameter is used for output.

Determines if a subscription with the specified *event_type_name* and *filter* exists on the Broker for the *txclient*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>event_type_name</i> or <i>exists</i> is NULL.

See also:

“awCancelTxSubscription” on page 158

“awNewTxSubscription” on page 306

awEnd

awEndTransaction

```
BrokerError awEndTransaction(
    BrokerClient client,
    char *transaction_id,
    int mode,
    int *reply_tag);
```

<i>client</i>	The Broker client that is beginning the transaction.
<i>transaction_id</i>	The identifier representing the transaction to be ended.
<i>mode</i>	Identifies the transaction mode and should be one of the following values: <ul style="list-style-type: none"> ■ TRANSACTION_MODE_COMMIT ■ TRANSACTION_MODE_ROLLBACK ■ TRANSACTION_MODE_SAVEPOINT
<i>reply_tag</i>	If not NULL, indicates that a reply is requested and the * tag value of this event is returned. This parameter is used for output.

Performs commit, rollback, or savepoint processing for a transaction by publishing an `Adapter::endTransaction` event for the given transaction identifier. If `reply_tag` is not `NULL`, it will be set with the tag value of the event published so that you can match it to the acknowledgment that will follow. If `reply_tag` is `NULL`, no acknowledgment reply will be sent. This function should be used in conjunction with the [awBeginTransaction](#) function.

Possible BrokerError major codes	Meaning
<code>AW_ERROR_INVALID_CLIENT</code>	The <i>client</i> parameter is not <code>NULL</code> and has been destroyed or disconnected.
<code>AW_ERROR_NO_PERMISSION</code>	The <i>client</i> does not have permission to publish the <code>Adapter::endTransaction</code> event type.
<code>AW_ERROR_NULL_PARAM</code>	The <i>transaction_id</i> parameter is <code>NULL</code> .
<code>AW_ERROR_OUT_OF_RANGE</code>	The <i>mode</i> parameter contains an illegal value.

See also:

“[awBeginTransaction](#)” on page 150

awError

awErrorToCompleteString

```
char * awErrorToCompleteString(
    BrokerError err);
```

err The error to be converted to a string.

Returns a string that describes the specified *err* in greater detail than the function `awErrorToString`. Caller is *not* responsible for freeing memory associated with the returned value.

Note:

The value is valid only until the next error is generated.

See also:

“[awErrorToString](#)” on page 190

awErrorToCompleteStringUC

```
charUC * awErrorToCompleteStringUC(BrokerError err);
```

Gets a unicode string describing the error that occurred but with more information. Caller is responsible for freeing the return value. Returns NULL if the parameter is not an error or if runs out of memory.

awErrorToString

```
char * awErrorToString(BrokerError err);
```

err The error to be converted to a string.

Returns a string that gives a brief description of the specified *err*. The caller is *not* responsible for freeing the memory associated with the returned value.

Note:

The value is valid only until the next error is generated.

See also:

“awErrorToCompleteString” on page 189

awErrorToStringUC

```
charUC * awErrorToStringUC(BrokerError err);
```

Gets a unicode string describing the error that occurred. Caller is responsible for freeing the return value. Returns NULL if the parameter is not an error or if runs out of memory.

awEvent

awEventFormatAssemble

```
BrokerError awEventFormatAssemble(  
    BrokerEvent event,  
    BrokerFormatToken *tokens,  
    char **result);
```

event The event to be formatted.

tokens The format string to use.

result The resulting format string. This parameter is used for output.

Assembles a string from the list of *tokens*, replacing field references with values from *event*. A pointer to the resulting string is returned in *result*, which the caller is responsible for freeing.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	A field referenced in the <i>tokens</i> does not exist in the event.
AW_ERROR_INVALID_TYPE	A field is not of a type supported by the string formatter.

See also:

“awEventFormatBindVariable” on page 191

“awEventFormatFree” on page 192

“awEventFormatPreparse” on page 192

“awEventFormatTokens” on page 193

“awEventToBinData” on page 194

“awSetFormatMode” on page 347

awEventFormatBindVariable

```
char * awEventFormatBindVariable(
    BrokerFormatToken *tokens,
    int index,
    char **placeholderName);
```

tokens The list of tokens.

index The index into the list of tokens.

placeholderName The resulting string. This parameter is used for output.

Returns the name of the bind variable at *token[index]*. Returns NULL if there is no bind variable at that position, or if the index is not valid. The caller should not modify the return value. If *placeholderName* is not zero, it will be set to a string that represents the place holder in the statement (for example, ":v0" or "?").

See also:

“awEventFormatAssemble” on page 190

“awEventFormatFree” on page 192

“awEventFormatPreparse” on page 192

“awEventFormatTokens” on page 193

“awEventToBinData” on page 194

“awSetFormatMode” on page 347

awEventFormatFree

```
void awEventFormatFree(  
    BrokerFormatToken *tokens);
```

tokens The tokens to be freed.

Frees the list of *tokens*.

See also:

“awEventFormatAssemble” on page 190

“awEventFormatBindVariable” on page 191

“awEventFormatPreparse” on page 192

“awEventFormatTokens” on page 193

“awEventToBinData” on page 194

“awSetFormatMode” on page 347

awEventFormatPreparse

```
BrokerFormatToken* awEventFormatPreparse(  
    char *format_string);
```

format_string The format string to use.

Breaks *format_string* into a list of tokens. The list is NULL terminated. The return value must be freed by the caller using `awEventFormatFree`. You may call this function once to create a list of tokens that may be used with several different events by calling the [awEventFormatAssemble](#) function.

See also:

“awEventFormatAssemble” on page 190

“awEventFormatBindVariable” on page 191

“awEventFormatFree” on page 192

“awEventFormatTokens” on page 193

“awEventToBinData” on page 194

“awSetFormatMode” on page 347

awEventFormatTokens

```
int awEventFormatTokens(
    BrokerFormatToken *tokens);
```

tokens The array of tokens to be counted.

Returns the number of tokens in the specified array.

See also:

“awEventFormatAssemble” on page 190

“awEventFormatBindVariable” on page 191

“awEventFormatFree” on page 192

“awEventToBinData” on page 194

“awSetFormatMode” on page 347

awEventFromBinData

```
BrokerError awEventFromBinData(
    BrokerClient client,
    char *data,
    int size,
    BrokerEvent *event);
```

client The client that is associated with this event. This may be set to NULL.

data The binary data to use to create the event.

size The size of the *data*.

event The area where the created event is returned. This parameter is used for output.

Creates a Broker event using a byte array previously returned by the [awEventToBinData](#) function. The *client* may be NULL if you wish to create the event without type checking its contents.

The caller is responsible for calling [awDeleteEvent](#) on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_CORRUPT	The <i>data</i> parameter does not point to a valid event.
AW_ERROR_INVALID_CLIENT	The <i>client</i> parameter is not NULL and has been destroyed or disconnected.

Possible BrokerError major codes	Meaning
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish or subscribe to <i>event_type_name</i> .
AW_ERROR_NULL_PARAM	The <i>event</i> or <i>data</i> parameter is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awEventToBinData” on page 194

awEventToBinData

```
BrokerError awEventToBinData(
    BrokerEvent event,
    char **data,
    int *size);
```

<i>event</i>	The event to be converted to a binary array.
<i>data</i>	The area where the binary data is to be written. This parameter is used for output.
<i>size</i>	The size of the binary data array that is created. This parameter is used for output.

Creates a binary array representation of the specified *event*. The binary *data* can be stored on disk for later retrieval and conversion back to a Broker event.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>event_type_name</i> or <i>exists</i> is NULL.

See also:

“awEventFromBinData” on page 193

awEventToFormattedString

```
BrokerError awEventToFormattedString(
    BrokerEvent event,
    char *format_string,
    char **result);
```

<i>event</i>	The event to be formatted.
<i>format_string</i>	The format string to use.
<i>result</i>	The resulting format string. This parameter is used for output.

Formats a string containing specific field values from *event*, based on the content of *format_string*. Lines in the resulting string are separated with newline characters '\n'. A pointer to the resulting string is returned in *result*, which the caller is responsible for freeing.

This function uses the C language locale for formatting event fields with a `float`, `double`, or `BrokerDate` data type.

This function returns an error if the event is invalid, if *format_string* or *result* are `NULL`, or if the string could not be formatted properly.

Several options are available for specifying the format of event fields in *format_string*.

Format option	Description
s	Encloses the output value in single quotes. Single quotes occurring in the value are escaped with a backslash ("\") character.
d	Encloses the output value in double quotes. Double quotes occurring in the value are escaped with a backslash ("\") character.
q	Provides SQL-style quoting. This is like the :s option, but the escape character is a single quote instead of slash.
v	Replaces each field reference with a bind variable place holder. The default style is :v1, :v2. You can change this option to use ? by calling the <code>awSetFormatMode</code> function.
c	Outputs a numeric value from a field containing a string. Any non-numeric portions of the string field will be discarded. "123x" will become "123".
<number>	Specifies a minimum field length together with one of the options above. The value of string field containing AB with a format option of 5d will result in the output "AB". The field is always left justified and right padded as necessary to meet the field length.

Event fields can be referenced in *format_string* as follows:

1. Field names must be preceded with a \$ character.
2. Field names may optionally be enclosed in curly braces.
3. Format options must be placed at the end of the field name and be separated with a : character.
4. If a minimum field width is specified, it must follow the : delimiter and precede the format specifier.

5. Use a slash to escape any reserved characters, such as the \$ character.

You could use either of the following formats to refer to a string field named age:

```
$age:c
```

or

```
${age:c}
```

See also:

“awEventFormatAssemble” on page 190

“awEventFormatBindVariable” on page 191

“awEventFormatFree” on page 192

“awEventToBinData” on page 194

“awEventToString” on page 196

awEventToLocalizedString

```
char * awEventToLocalizedString(  
    BrokerEvent event);
```

event The event to be converted to a string.

Uses the current locale to convert all the fields in the specified *event* into a string that is suitable for display. Lines in the resulting string are separated with `newline` characters (`\n`). The caller is responsible for calling `free` on the returned value.

See also:

“awEventToBinData” on page 194

“awEventToString” on page 196

awEventToString

```
char * awEventToString(  
    BrokerEvent event);
```

event The event to be converted to a string.

Converts the fields of the specified *event* to a string that is suitable for display. Lines in the resulting string are separated with `newline` characters (`\n`). The caller is responsible for calling `free` on the returned value.

This function uses the C language locale for formatting event fields with a `float` or `double` data type. Fields containing dates are formatting using the following C language-like format:

```
mm/dd/yyyy hh.mm.ss[.ms]
```

See also:

“`awEventToBinData`” on page 194

“`awEventToLocalizedString`” on page 196

awFilter

awFilterToString

```
char * awFilterToString(  
    BrokerFilter filter);
```

filter The filter to be converted to a string.

Returns the *filter* as a string. The caller is responsible for freeing the memory associated with the return value.

This function returns `NULL` if *filter* is invalid or if memory cannot be allocated.

awFree

awFree

```
void awFree(  
    void *ptr);
```

ptr The memory to be freed.

Frees the memory pointed to by *ptr*. Guaranteed to use the same heap as the `webMethods Broker C` library.

Important:

You must use this function to free memory allocated by the `webMethods Broker C` library functions for applications on Win32 platforms when you intend to turn on memory debugging.

See also:

“`awMalloc`” on page 289

“`awRealloc`” on page 319

awFlush

awFlushTypeDefCache

```
BrokerError awFlushTypeDefCache(
    BrokerClient client);
```

client The Broker client whose event type definition cache is to be flushed.

Flushes all type definitions from the cache associated with the specified *client*. See [“Event Type Definition Cache” on page 117](#) for more information on using this function.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.

See also:

“awLockTypeDefCache” on page 284

“awUnlockTypeDefCache” on page 394

awGet

awGet<type>Field

Functions such as `awGetBooleanField` and `awGetShortField` have the following general syntax:

```
BrokerError awGet<type>Field(
    BrokerEvent event,
    char *field_name,
    <Field_value_type> value);
```

event The event whose field is to be returned.

field_name The name of the field to be returned. Note that *field_name* can directly identify any field in the event, no matter how deeply nested that field may be.

value The value associated with the field that is output from this function.

The following functions are provided to return the value associated with a specific field type.

Function Name	Field Value Type
<code>awGetBooleanField</code>	<code>BrokerBoolean *</code>

Function Name	Field Value Type
awGetByteField	char *
awGetCharField	char *
awGetDateField	BrokerDate *
awGetDoubleField	double *
awGetFloatField	float *
awGetIntegerField	int *
awGetLongField	BrokerLong *
	Note: Available only on platforms which support 64-bit integral representations.
awGetLongFieldNative	NativeLong *
awGetShortField	short *
awGetStringField	char **
	Note: The caller is responsible for calling free on the value.
awGetUCCharField	charUC *
awGetUCStringField	charUC **b

See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

If you get a field whose value was not set by the event's publisher, the field will contain one of the following default values:

- Boolean values will have a value of 0 (false).
- Integral data types (such as int, long, and short) will have a value of zero.
- Floating point types will have a value of 0.0.
- String types will have a contain a zero-length string.
- BrokerDate types will be empty. See [awNewEmptyBrokerDate](#) for more information.

Note:
When awGetStringField is used to obtain the value of a field whose type is FIELD_TYPE_UNICODE_STRING, the value returned will automatically be converted to an ANSI string.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event or an attempt was made to obtain the value of an envelope field that has not been set.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> or <i>value</i> is NULL.

See also:

“awGet<type>SeqField” on page 200

“awGetField” on page 241

“awGetSequenceField” on page 248

“awGetStructFieldAsEvent” on page 252

“awGetStructSeqFieldAsEvents” on page 253

“awGetUCStringFieldAsA” on page 276

“awGetUCStringFieldAsUTF8” on page 277

awGet<type>SeqField

Functions such as `awGetBooleanSeqField` and `awGetShortSeqField` have the following general syntax:

```
BrokerError awGet<type>SeqField(
    BrokerEvent event,
    char *field_name,
    int offset,
    int max_n,
    int *n,
    <Field_Value_Type> value_array);
```

<i>event</i>	The event whose field is to be returned.
<i>field_name</i>	The name of the sequence field to be returned from the event.
<i>offset</i>	The number of elements to skip from the beginning of the sequence.
<i>max_n</i>	The number of elements requested. If set to -1, all elements are requested.

<i>n</i>	The number of elements actually returned. This is an output value set by the function.
<i>value_array</i>	An array of values associated with the field that is output from this function.

The following functions are provided to return the values associated with a specific sequence field type.

Function Name	Field Value Type
awGetBooleanSeqField	BrokerBoolean **
awGetByteSeqField	char **
awGetCharSeqField	char **
awGetDateSeqField	BrokerDate **
awGetDoubleSeqField	double **
awGetFloatSeqField	float **
awGetIntegerSeqField	int **
awGetLongSeqField	BrokerLong **
awGetShortSeqField	short **
awGetStringSeqField	char ***
awGetUCCharSeqField	charUC **
awGetUCStringSeqField	charUC ***

These functions return the values of sequence-type event fields, as listed above. Values of event fields can be requested in any order. If you get a sequence field whose value was not set by the event's publisher, a zero-length sequence will be returned.

To get a sequence of structures, use the [awGetStructSeqFieldAsEvents](#) function. To get values of non-sequence type event fields, use one of the `awGet<type>Field`.

A *sequence* is a series of values with the same type, similar in concept to an array. Sequence-type event fields are created using one of the [awSet<type>SeqField](#) functions. You can create sequences of more than one dimension by building sequences of sequences. To get all sequence values in one call, set *max_n* = -1 and *offset* = 0.

You may limit the number of sequence values returned by specifying a *max_n* $\setminus > 0$ and *offset* = 0 on the first call. For subsequent calls, set *offset* to the number of elements already processed, so that the elements already retrieved may be skipped. For example, if there are 75 strings in a sequence, and you want to handle no more than 50 on each function call:

- On the first call to `awGetStringSeqField`, pass:

max_n=50, offset=0, n=50 (returned)

- On the second call to `awGetStringSeqField`, pass:

max_n=50, offset=50 (already processed),

n=25 (returned)

- On return, every function returns a pointer to an array; for example:

- The `awGetCharSeqField` function returns a pointer to an array of chars.

- The `awGetStringSeqField` function returns a pointer to an array of string pointers.

Note:

When `awGetStringSequenceField` is used to obtain the value of a sequence field whose type is `FIELD_TYPE_UNICODE_STRING`, the values returned will automatically be converted to ANSI strings.

The caller is responsible for calling `free` on *value_array*, which is allocated in one block of storage. See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

Possible BrokerError major codes	Meaning
<code>AW_ERROR_FIELD_NOT_FOUND</code>	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event or an attempt was made to obtain the value of an envelope field that has not been set.
<code>AW_ERROR_FIELD_TYPE_MISMATCH</code>	The field's type does not match the type of <i>value_array</i> or the <i>field_name</i> incorrectly accesses a type.
<code>AW_ERROR_INVALID_EVENT</code>	The <i>event</i> is invalid.
<code>AW_ERROR_INVALID_FIELD_NAME</code>	The <i>field_name</i> is invalid.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>field_name</i> , <i>n</i> , or <i>value</i> is NULL.
<code>AW_ERROR_OUT_OF_RANGE</code>	The <i>offset</i> is less than zero or greater than the sequence size. Also returned if <i>max_n</i> is less than zero.

See also:

“`awGet<type>Field`” on page 198

“`awGetField`” on page 241

“`awGetSequenceField`” on page 248

“`awGetStructFieldAsEvent`” on page 252

“`awGetStructSeqFieldAsEvents`” on page 253

“`awGetUCStringSeqFieldAsUTF8`” on page 278

awGetBooleanField

See [awGet<type>Field](#).

awGetBooleanSeqField

See [awGet<type>SeqField](#).

awGetBrokerLong

```
void awGetBrokerLong(
    BrokerLong *bl,
    int *high,
    int *low);
```

bl The `BrokerLong` whose high and low-order words are to be obtained.

high The high-order word. This parameter is used for output.

low The low-order word. This parameter is used for output.

Obtains the high and low-order words from the specified `BrokerLong`.

The parameter *high* will be set to the most significant 32 bits. The parameter *low* will be set to the least significant 32 bits.

See also:

“[awBrokerToNativeLong](#)” on page 152

“[awGetLongFieldNative](#)” on page 245

“[awNativetoBrokerLong](#)” on page 290

“[awSetLongFieldNative](#)” on page 348

awGetBrokerSSLCertificate

```
BrokerError awGetBrokerSSLCertificate(
    BrokerClient client,
    BrokerSSLCertificate **certificate);
```

client The Broker client whose Broker's SSL certificate is to be returned.

certificate The Broker's certificate. This parameter is used for output.

Obtains the SSL certificate for the Broker to which this *client* is connected.

The caller is responsible for freeing the memory associated with the output *certificate*. The `BrokerSSLCertificate` structure is allocated as one block of memory, so nested freeing is unnecessary.

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_BAD_STATE</code>	The <i>client</i> is not using an SSL connection.
<code>AW_ERROR_INVALID_CLIENT</code>	The <i>client</i> has been destroyed or disconnected.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>certificate</i> is NULL.

awGetBrokerVersionNumber

```
BrokerError awGetBrokerVersionNumber(
    BrokerClient client,
    int *version);
```

client The Broker client whose Broker version number is to be returned.

version The version number, returned as one of the `AW_VERSION_*` values. This parameter is used for output.

Obtains the version number of the Broker to which this *client* is connected. If the Broker's version number is newer than the client's version, the client's version number is returned.

The caller is responsible for freeing the memory associated with the output value.

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_INVALID_CLIENT</code>	The <i>client</i> is not valid.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>version</i> is NULL.

awGetByteField

See [awGet<type>Field](#).

awGetByteSeqField

See [awGet<type>SeqField](#).

awGetCanPublishNames

```
BrokerError awGetCanPublishNames(
    BrokerClient client,
    int *n,
    char ***event_type_names);
```

<i>client</i>	The Broker client whose list of event types is to be returned.
<i>n</i>	The number of event type names that were returned. This parameter is used for output.
<i>event_type_names</i>	The array of event type names that this Broker client can publish. This parameter is used for output.

Obtains the name of every event type that the specified *client* can publish or deliver to the Broker client's Broker. The caller is responsible for calling *free* on *event_type_names*. All strings in *event_type_names* share one memory block so nested freeing is unnecessary.

The events that a client may publish are determined by the client group to which the Broker client belongs. For information on client groups, see *Administering webMethods Broker*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> or <i>event_type_names</i> is NULL.

See also:

“awGetCanPublishTypeDefs” on page 205

“awGetCanSubscribeNames” on page 206

“awGetCanSubscribeTypeDefs” on page 207

awGetCanPublishTypeDefs

```
BrokerError awGetCanPublishTypeDefs(
    BrokerClient client,
    int *n,
    BrokerTypeDef **defs);
```

<i>client</i>	The Broker client whose list of event type definitions is to be returned.
<i>n</i>	The number of event type definitions that were returned. This parameter is used for output.
<i>defs</i>	The array of event type definitions that this Broker client can publish. This parameter is used for output.

Returns the definitions for the event types that this Broker client can publish. The caller is responsible for calling *free* on the output array, but not on the event type definitions themselves.

The events that a client may publish are determined by the client group to which the Broker client belongs. For information on client groups, see *Administering webMethods Broker*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> or <i>defs</i> is NULL.

See also:

“awGetCanPublishNames” on page 204

“awGetCanSubscribeNames” on page 206

“awGetCanSubscribeTypeDefs” on page 207

awGetCanSubscribeNames

```
BrokerError awGetCanSubscribeNames(
    BrokerClient client,
    int *n,
    char ***event_type_names);
```

<i>client</i>	The Broker client whose list of event types is to be returned.
<i>n</i>	The number of event type names returned. This parameter is used for output.
<i>event_type_names</i>	The array of event type names to which this Broker client can subscribe. This parameter is used for output.

Returns the name of every event type known to the Broker to which the *client* can subscribe. These are also the event types that are permissible to be delivered to the *client*. The caller is responsible for calling `free` on *event_type_names*. All of the strings in *event_type_names* share one memory block so nested calls to `free` are not necessary.

The events to which a client may subscribe is determined by the client group to which the Broker client belongs. For information on client groups, see *Administering webMethods Broker*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> or <i>event_type_names</i> is NULL.

See also:

“awGetCanPublishTypeDefs” on page 205

“awGetCanSubscribeNames” on page 206

“awGetCanSubscribeTypeDefs” on page 207

awGetCanSubscribeTypeDefs

```
BrokerError awGetCanSubscribeTypeDefs(
    BrokerClient client,
    int *n,
    BrokerTypeDef **defs);
```

<i>client</i>	The Broker client whose list of event type definitions is to be returned.
<i>n</i>	The number of event type definitions that were returned. This parameter is used for output.
<i>def</i>	The array of event type definition to which this Broker client can subscribe. This parameter is used for output.

Provides the definitions for the event types to which this Broker client can subscribe. These are also the event types that are permissible to be delivered to the *client*. The caller is responsible for calling `free` on the *defs* array, but not on the definitions themselves.

The events to which a client may subscribe is determined by the client group to which the Broker client belongs. For information on client groups, see *Administering webMethods Broker*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> or <i>defs</i> is NULL.

See also:

- “awGetCanPublishNames” on page 204
- “awGetCanSubscribeNames” on page 206
- “awGetCanSubscribeTypeDefs” on page 207

awGetCharField

See [awGet<type>Field](#).

awGetCharSeqField

See [awGet<type>SeqField](#).

awGetClientAccessLabel

```
BrokerError awGetClientAccessLabel(
    BrokerClient client,
```

```
int *n,
short **label);
```

<i>client</i>	The Broker client whose list of event types is to be returned.
<i>n</i>	The number of element returned in the <i>label</i> array. This parameter is used for output.
<i>label</i>	The access label for the client. This parameter is used for output.

Obtains the access label for the specified client. This label will be inserted in the `pubLabel` envelope field of every event the client publishes or delivers.

Access labels provide access control for event types which is independent of the event type.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	The <i>client</i> has an owner, but the access label feature is not enabled.
AW_ERROR_BROKER_FAILURE	The <i>client</i> has an owner, but an error occurred looking up the access label.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have an owner so it cannot have an access label.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> or <i>label</i> is NULL.

See also:

“`awSetClientAutomaticControlLabel`” on page 332

awGetClientApplicationName

```
BrokerError awGetClientApplicationName(
    BrokerClient client,
    char **app_name);
```

<i>client</i>	The Broker client whose list of event types is to be returned.
<i>app_name</i>	The name of the application associated with this Broker client. This parameter is used for output.

Provides the application name for this Broker client. The caller is responsible for freeing the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>app_name</i> is NULL.

See also:

“awNewBrokerClient” on page 291

“awReconnectBrokerClient” on page 319

awGetClientBrokerHost

```
BrokerError awGetClientBrokerHost(
    BrokerClient client,
    char **host_name);
```

client The Broker client whose Broker host name is to be returned.

host_name The name of the Broker host associated with this Broker client. This parameter is used for output.

Provides a string containing the host name of the Broker for this Broker client. The caller is responsible for calling `free` on the *host_name* string.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> not valid.
AW_ERROR_NULL_PARAM	The parameter <i>host_name</i> is NULL.

See also:

“awGetClientBrokerName” on page 209

“awGetClientBrokerPort” on page 210

“awNewBrokerClient” on page 291

“awReconnectBrokerClient” on page 319

awGetClientBrokerName

```
BrokerError awGetClientBrokerName(
    BrokerClient client,
    char **broker_name);
```

<i>client</i>	The Broker client whose Broker name is to be returned.
<i>broker_name</i>	The Broker name associated with this Broker client. This parameter is used for output.

Provides a string containing the Broker name for this Broker client. The caller is responsible for calling `free` on the *broker_name* string.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>broker_name</i> is NULL.

See also:

“`awGetClientBrokerHost`” on page 209

“`awGetClientBrokerPort`” on page 210

“`awNewBrokerClient`” on page 291

“`awReconnectBrokerClient`” on page 319

awGetClientBrokerPort

```
BrokerError awGetClientBrokerPort(
    BrokerClient client,
    int *port_number);
```

<i>client</i>	The Broker client whose client Broker port is to be returned.
<i>port_number</i>	The port number of the Broker Server. This parameter is used for output.

Provides a int value containing the port number of the Broker Server.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>port_number</i> is NULL.

See also:

“`awGetClientBrokerHost`” on page 209

“`awGetClientBrokerName`” on page 209

“`awNewBrokerClient`” on page 291

“awReconnectBrokerClient” on page 319

awGetClientConnectionDescriptor

```
BrokerError awGetClientConnectionDescriptor(
    BrokerClient client,
    BrokerConnectionDescriptor *desc);
```

client The Broker client whose connection descriptor is to be returned.

desc The Broker client's connection descriptor. This parameter is used for output.

Obtains the connection descriptor for *client* and returns it in *desc*. If a NULL descriptor argument was passed to `awNewBrokerClient` when *client* was created, then a descriptor which matches the default connection descriptor behavior is retrieved.

Note:

This function cannot be used to get the actual Broker settings for a connection descriptor.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>desc</i> is NULL.

See also:

“awNewBrokerClient” on page 291

“awNewBrokerConnectionDescriptor” on page 292

“awReconnectBrokerClient” on page 319

awGetClientGroup

```
BrokerError awGetClientGroup(
    BrokerClient client,
    char **group_name);
```

client The Broker client whose client group name is to be returned.

group_name The name of the client group to which this Broker client belongs. This parameter is used for output.

Creates a string containing the client group name. The caller is responsible for calling `free` on the *group_name* string. For more information on client groups, see *Administering webMethods Broker*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>group_name</i> is NULL.

See also:

“awNewBrokerClient” on page 291

“awReconnectBrokerClient” on page 319

awGetClientId

```
BrokerError awGetClientId(
    BrokerClient client,
    char **client_id);
```

client The Broker client whose identifier is to be returned.

client_id The client id that is associated with this Broker client. This parameter is used for output.

Creates a string containing the client identifier. The caller is responsible for calling *free* on the *client_id* string.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>client_id</i> is NULL.

See also:

“awNewBrokerClient” on page 291

“awReconnectBrokerClient” on page 319

awGetClientInfoSet

```
BrokerError awGetClientInfoSet(
    BrokerClient client,
    BrokerEvent *infoSet);
```

client The Broker client whose infoSet is to be returned.

infoSet The area where the retrieved infoSet is returned. This parameter is used for output.

Obtains the info set for the specified *client*. The info set is returned as a Broker event for convenience. Each client can store one info set that can be used to contain information about its state or configuration.

The caller is responsible for invoking [awDeleteEvent](#) on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>info set</i> is NULL.

See also:

“awSetClientInfo set” on page 332

awGetClientLastPublishSequenceNumber

```
BrokerError awGetClientLastPublishSequenceNumber(
    BrokerClient client,
    BrokerLong *seqn);
```

client The Broker client whose last publish sequence number is to be returned.

seqn The highest sequence number of the events published by this Broker client. This parameter is used for output.

Provides the highest sequence number used by the specified client in a publish or deliver call.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>seqn</i> is NULL.

See also:

“awPublishEvent” on page 314

“awSetEventPublishSequenceNumber” on page 344

awGetClientQueueLength

```
BrokerError awGetClientQueueLength(
    BrokerClient client,
    int *n);
```

client The Broker client whose queue length is to be returned.

n The number of events currently in the Broker client's event queue. This parameter is used for output.

Provides the number of events currently in event queue for *client*.

Note:

The length returned will include any unacknowledged events in the queue. This means the length may be greater than you expect if your Broker client has received events, but has not yet acknowledged them.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> is NULL.

See also:

“awCanPublish” on page 153

awGetClientSSLBrokerDistinguishedName

```
BrokerError awGetClientSSLBrokerDistinguishedName(
    BrokerClient client,
    char **distinguished_name);
```

Get the secure socket distinguished name for the Broker. The caller is responsible for freeing the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	The client is not using a secure socket connection.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> is NULL.

awGetClientSSLBrokerIssuerDistinguishedName

```
BrokerError awGetClientSSLBrokerIssuerDistinguishedName(
    BrokerClient client,
    char **distinguished_name);
```

Get the secure socket distinguished name of the issuer of the Broker's certificate. The caller is responsible for freeing the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	The client is not using a secure socket connection.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> is NULL.

awGetClientSSLEncryptionLevel

```
BrokerError awGetClientSSLEncryptionLevel(
    BrokerClient client,
    int *level);
```

client The Broker client whose SSL encryption level is to be returned.

level The SSL level being used. See the following table for a list of possible values. This parameter is used for output.

Returns an indication of the level of SSL that is supported as one of the values shown below. If SSL is not being used by this client, a `AW_BAD_STATE` exception is thrown.

SSL Level	Description
AW_SSL_LEVEL_NO_ENCRYPTION	Encryption support is not available.
AW_SSL_LEVEL_US_EXPORT	US export level encryption is supported, which limits the size of the keys used.
AW_SSL_LEVEL_US_DOMESTIC	US domestic level encryption is supported, which allows for larger key sizes and greater security.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	The <i>client</i> is not using SSL.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>level</i> is NULL.

See also:

“`awGetSSLEncryptionLevel`” on page 251

awGetClientStateShareLimit

```
BrokerError awGetClientStateShareLimit(
    BrokerClient client,
    int *limit);
```

<i>client</i>	The Broker client whose state share limit is to be returned.
<i>limit</i>	The maximum number of clients that may share this Broker client's state. This parameter is used for output.

Provides the maximum number of Broker clients that can share *client*'s state. The client state includes the Broker client's event queue and list of subscribed events. If the limit returned is -1, this indicates there is no limit to the number of Broker clients that can share this state.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>limit</i> is NULL.

See also:

“awSetCharField” on page 332

“awSetDescriptorSSLEncrypted” on page 342

awGetClientTerritoryName

```
BrokerError awGetClientTerritoryName(  
    BrokerClient client,  
    char **territory_name);
```

<i>client</i>	The Broker client whose Broker's territory name is to be returned.
<i>territory_name</i>	The area where the name is to be returned. This parameter is used for output.

Returns the territory name associated with the Broker to which the *client* is connected.

The caller is responsible for calling `free` on the returned value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>territory_name</i> is NULL.

See also:

“awGetTypeDefTerritoryName” on page 274

awGetDateCtime

```
BrokerError awGetDateCtime(
    BrokerDate *d,
    time_t *t);
```

- d* The Broker date from which the time is to be extracted.
- t* The client id that is associated with this Broker client. This parameter is used for output.

Obtains a `time_t` value from the specified Broker date. See C date and time functions in `<time.h>` for information on `time_t`.

Possible BrokerError major codes	Meaning
AW_ERROR_OUT_OF_RANGE	The specified date, <i>d</i> , cannot be represented as a <code>time_t</code> date. A date earlier than January 1, 1970 or after 03:14:07 on January 19, 2038 cannot be represented as a <code>time_t</code> date.

Important:

Millisecond accuracy is lost when a `BrokerDate` is converted to the `time_t` format.

See also:

“[awSetDateCtime](#)” on page 334

awGetDateField

See [awGet<type>Field](#).

awGetDateSeqField

See [awGet<type>SeqField](#).

awGetDefaultBrokerPort

```
int awGetDefaultBrokerPort();
```

Returns the default number for connecting to Brokers. The default port is used for non-SSL connections and has a value of 6849. The default port number is also used to calculate the port numbers shown in “[Basic Properties](#)” on page 50.

See also:

“awSetDefaultClientTimeout” on page 336

awGetDescriptorAccessLabelHint

```
BrokerError awGetDescriptorAccessLabelHint(
    BrokerConnectionDescriptor desc,
    char **hint);
```

desc The Broker descriptor. This parameter is used for output.

hint The hint string to use when looking up a client's access label. This parameter is used for output.

Returns the access label hint defined for this connection descriptor.

When using the access label feature, a client may specify a hint string to be used to look up the client's access label. The access label look-up occurs only when creating a client. The hint string will be ignored when reconnecting a Broker client.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.

See also:

“awSetDescriptorAccessLabelHint” on page 336

awGetDescriptorAuthUserName

```
BrokerError awGetDescriptorAuthUserName(
    BrokerConnectionDescriptor desc,
    char **user_name);
```

desc The Broker descriptor. This parameter is used for output.

user_name The user name available in the descriptor, *desc*.

Returns the user name of the basic authentication user available in the descriptor.

Note:
You must free the output *user_name* string after a successful `awGetDescriptorAuthUserName` call.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAM	The <i>user_name</i> parameter is NULL.
AW_ERROR_NO_MEMORY	Cannot allocate memory for the output parameter, <i>user_name</i> .

See also:

“awSetDescriptorAuthInfo” on page 336

awGetDescriptorAutomaticReconnect

```
BrokerError awGetDescriptorAutomaticReconnect(
    BrokerConnectionDescriptor desc,
    BrokerBoolean *reconnect);
```

<i>desc</i>	The Broker descriptor. This parameter is used for output.
<i>reconnect</i>	Indicates whether or not the automatic reconnect behavior is to be enabled for this descriptor. This parameter is used for output.

Returns 1 (*true*) if the automatic reconnection feature for this descriptor is enabled. Otherwise, 0 (*false*) is returned.

Note:

You can disable the automatic reconnection feature by explicitly invoking `awDisconnectClient` or `awDestroyClient`.

If automatic reconnection is enabled, the Broker client associated with this descriptor will be automatically reconnected if the connection to the Broker is lost. See [“Automatic Reconnection” on page 54](#) for a complete discussion of the automatic reconnect feature.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.
AW_ERROR_NULL_PARAM	The <i>reconnect</i> parameter is NULL.

See also:

“awSetDescriptorAutomaticReconnect” on page 337

awGetDescriptorAutomaticRedeliveryCount

```
BrokerError awGetDescriptorAutomaticRedeliveryCount(
    BrokerConnectionDescriptor desc,
    BrokerBoolean *auto_on);
```

desc The Broker descriptor for which the automatic redelivery count status is to be returned.

auto_on Indicates whether or not the automatic redelivery count is enabled for this descriptor. This parameter is used for output.

Returns 1 (true) if automatic redelivery count is enabled. Otherwise, 0 (false) is returned.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter s invalid.
AW_ERROR_NULL_PARAM	The <i>auto_on</i> parameter is NULL.

See also:

“awSetDescriptorAutomaticRedeliveryCount” on page 338

awGetDescriptorConnectionShare

```
BrokerError awGetDescriptorConnectionShare(  
    BrokerConnectionDescriptor desc,  
    BrokerBoolean *shared);
```

desc The Broker descriptor. This parameter is used for output.

shared Indicates whether or not the specified connection to the Broker can be shared. This parameter is used for output.

Provides the value of the connection share attribute for the descriptor *desc*. If *shared* is 0 (false), then *desc* can only be used by one client.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>shared</i> is NULL.

See also:

“awSetDescriptorConnectionShare” on page 338

awGetDescriptorForcedReconnect

```
BrokerError awGetDescriptorForcedReconnect(  
    BrokerConnectionDescriptor desc,  
    BrokerBoolean*forced_reconnect);
```

<i>desc</i>	The Broker descriptor for which the forced reconnect status is to be returned.
<i>forced_reconnect</i>	Indicates whether a client program can forcibly reconnect to a Broker. Returns 1 (true) if forced reconnect is enabled for this descriptor. Otherwise, returns 0 (false).

Returns 1 (true) if forced reconnect is enabled for this descriptor. Otherwise, returns 0 (false).

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.
AW_ERROR_NULL_PARAM	The <i>forced_reconnect</i> parameter is NULL.

See also:

“awSetDescriptorForcedReconnect” on page 339

awGetDescriptorKeepAlive

```
BrokerError awGetDescriptorKeepAlive(
    BrokerConnectionDescriptor desc,
    int* KeepAliveSendPeriod,
    int* MaxKeepAliveResponseTime,
    int* RetryCount);
```

<i>desc</i>	The Broker descriptor for which the keep-alive status is to be returned.
<i>KeepAliveSendPeriod</i>	The length of time (in seconds) the Broker waits before issuing the keep-alive messages on an idle connection.
<i>MaxKeepAliveResponseTime</i>	The length of time (in seconds) the Broker waits for a reply from an idle connection.
<i>RetryCount</i>	Number of times keep-alive messages are sent before disconnecting the unresponsive client.

Gets the keep-alive settings for the specified descriptor.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.

See also:

“awSetDescriptorKeepAlive” on page 339

awGetDescriptorRedeliveryCountEnabled

```
BrokerError awGetDescriptorRedeliveryCountEnabled(
    BrokerConnectionDescriptor desc,
    BrokerBoolean *redelivery_count);
```

desc The Broker descriptor for which the redelivery count status is to be returned.set.

redelivery_count Returns true if redelivery counting is enabled. Returns false if redelivery counting is disabled.

Returns the redelivery counting status for the specified descriptor.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.
AW_ERROR_NULL_PARAM	The <i>redelivery_count</i> parameter is NULL.

See also:

“awSetDescriptorRedeliveryCountEnabled” on page 340

awGetDescriptorSharedEventOrdering

```
BrokerError awGetDescriptorSharedEventOrdering(
    BrokerConnectionDescriptor desc,
    char **ordering);
```

desc The Broker descriptor whose shared event ordering is to be returned.

ordering The current shared event ordering state, which will be set to either AW_SHARED_ORDER_NONE or AW_SHARED_ORDER_BY_PUBLISHER. This parameter is used for output.

Obtains the shared event ordering status for the specified descriptor. See [“By-Publisher Event Ordering” on page 56](#) for more information.

The caller is responsible for freeing the memory associated with the returned ordering state.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>ordering</i> is NULL.

See also:

“awSetDescriptorSharedEventOrdering” on page 340

awGetDescriptorSSLCertificate

```
BrokerError awGetDescriptorSSLCertificate(
    BrokerConnectionDescriptor desc,
    char **certificate_file,
    char **trust_file);
```

<i>desc</i>	The connection descriptor whose SSL information is to be provided.
<i>certificate_file</i>	The certificate file associated with <i>desc</i> . This parameter is used for output. If a NULL value is set on return, then SSL is not in use for <i>desc</i> .
<i>trust_file</i>	The trust file associated with <i>desc</i> . This parameter is used for output. If a NULL value is set on return and <i>certificate_file</i> is not NULL, then only server-side authentication is in use for <i>desc</i> .

Provides the secure socket layer (SSL) *certificate_file* and *trust_file* for *desc*. The password associated with *desc* is not returned.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.
AW_ERROR_NULL_PARAM	The <i>certificate_file</i> or <i>trust_file</i> is NULL.

The caller is responsible for freeing the two output parameters, if non-NULL values are returned.

See also:

“awSetDescriptorSSLCertificate” on page 341

awGetDescriptorSSLEncrypted

```
BrokerError awGetDescriptorSSLEncrypted(
    BrokerConnectionDescriptor desc,
    BrokerBoolean *encrypted);
```

<i>desc</i>	The Broker descriptor whose SSL encryption state is to be returned.
<i>encrypted</i>	If 0 (<i>false</i>), only SSL handshaking is used. If 1 (<i>true</i>), all traffic on this descriptor is encrypted. This parameter is used for output.

Provides the value of the SSL encrypt flag for the descriptor *desc*. If *encrypted* is 0 (*false*), then data traffic for the descriptor will not be encrypted once the SSL handshaking between server and client has been completed. If 1 (*true*), all data traffic that occurs after SSL handshaking will be encrypted.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>encrypted</i> is NULL.

See also:

“awSetDescriptorSSLEncrypted” on page 342

awGetDescriptorStateShare

```
BrokerError awGetDescriptorStateShare(
    BrokerConnectionDescriptor desc,
    BrokerBoolean *shared);
```

<i>desc</i>	The connection descriptor to be checked.
<i>shared</i>	Determines whether or not this Broker client can share its client state on the Broker. This parameter is used for output.

Provides the value of the event queue sharing attribute for the descriptor *desc*. If *shared* is 0 (*false*), then the client state associated with *desc* can only be used by one client. If *shared* is 1 (*true*), the client state can be shared with other Broker clients.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>shared</i> is NULL.

See also:

“awGetClientStateShareLimit” on page 215

“awSetCharField” on page 332

“awSetDescriptorSSLEncrypted” on page 342

awGetDiagnostics

```
int awGetDiagnostics();
```

Returns a value that represents the system diagnostics level. The possible return values and their meanings are:

Diagnostic Level	Description
0	No error output will be produced.
1	Produces error output for major errors only. This is the default setting
2	Produces all error output for all methods.

See also:

[“awSetDiagnostics” on page 343](#)

awGetDoubleField

See [awGet<type>Field](#).

awGetDoubleSeqField

See [awGet<type>SeqField](#).

awGetErrorCode

```
int awGetErrorCode(
    BrokerError err);
```

err The error whose error code is to be returned.

Returns an error code that describes the problem associated with the specified *err*. Returns zero if there is no error.

See [“Managing Event Types” on page 113](#) for a complete list of error codes and their meanings.

See also:

[“awGetErrorMinorCode” on page 225](#)

awGetErrorMinorCode

```
int awGetErrorMinorCode(
    BrokerError err);
```

err The error whose minor code is to be returned.

Gets the minor code of an error.

See also:

“awGetErrorCode” on page 225

awGetEvent

```
BrokerError awGetEvent(
    BrokerClient client,
    int msec,
    BrokerEvent *event);
```

<i>client</i>	The Broker client whose next event is to be obtained.
<i>msec</i>	Number of milliseconds to wait for an event before timing out. If set to AW_INFINITE, this function will wait indefinitely.
<i>event</i>	The event that is returned. This parameter is used for output.

Acknowledges all previously retrieved events for *client*, then obtains a single event for *client*, if available. If no events are currently available, this function will wait for the number of milliseconds specified by *msec*. If the wait time expires, this function returns with error code AW_ERROR_TIMEOUT. Any *event* that is obtained may be one for which the *client* has registered a subscription, or it may be a delivered event.

Note:

Before exiting, Broker clients with an explicit-destroy life cycle that are using `awGetEvent` should explicitly acknowledge the receipt sequence number of the last event received using either [awAcknowledge](#) or [awAcknowledgeThrough](#). Failure to do so will result in the last event being received again the next time you connect the Broker client.

Using this function on a client that has registered callback functions will temporarily disable the callback mechanism for this Broker client until this function returns.

The caller is responsible for calling `awDeleteEvent` for the output event.

Note:

The Broker will delete all guaranteed events from the event queue once they are acknowledged. If you wish to receive an event without acknowledging any previously retrieved events, use the [awGetEventsWithAck](#) function and specify a sequence number of -1.

Possible BrokerError major codes	Meaning
AW_ERROR_INTERRUPTED	The awInterruptGetEvents function was invoked.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>msec</i> parameter is less than -1.
AW_ERROR_TIMEOUT	The <i>msec</i> time-out interval expired before an event arrived.

See also:

“awDispatch” on page 186

“awGetEvents” on page 230

“awGetEventsWithAck” on page 231

“awInterruptGetEvents” on page 279

“awMainLoop” on page 284

“awThreadedCallbacks” on page 371

awGetEventClient

```
BrokerError awGetEventClient(
    BrokerEvent event,
    BrokerClient *client);
```

event The event whose associated client is to be returned.

client The Broker client associated with the specified event. This is an output parameter.

Obtains the *client* for this *event*. The output value is NULL if there is no *client* associated with the *event*. This function returns an error if the *event* is invalid.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>client</i> is NULL.

See also:

“awGetField” on page 241

“awSetField” on page 345

awGetEventFieldType

```
BrokerError awGetEventFieldType(
    BrokerEvent event,
    char *field_name,
    short *field_type);
```

event The event containing the desired field type.

<i>field_name</i>	The name of the field within the event.
<i>field_type</i>	The returned field type, such as <code>FIELD_TYPE_BOOLEAN</code> , defined in <code><awetdef.h></code> . This parameter is used for output.

Provides the field type for the event field with the specified *field_name*. The *field_name* may directly identify any field in an event, no matter how deeply nested.

If you request type information for a sequence field named *mySeq*, this function will return `FIELD_TYPE_SEQUENCE`. If you want to obtain the type of the elements contained in *mySeq*, you can use this method with the field name *mySeq[]*. See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

You may obtain field type information about simple sequence fields in events created without a Broker client. However, you cannot obtain field type information on a field contained within sequence field or within a structure field unless the event was created with a Broker client.

Possible BrokerError major codes	Meaning
<code>AW_ERROR_FIELD_NOT_FOUND</code>	The <i>field_name</i> does not exist in the event.
<code>AW_ERROR_FIELD_TYPE_MISMATCH</code>	The <i>field_name</i> incorrectly accesses a type, such as using a subscript on a non-sequence field.
<code>AW_ERROR_INVALID_EVENT</code>	The <i>event</i> is invalid.
<code>AW_ERROR_INVALID_FIELD_NAME</code>	The <i>field_name</i> is invalid.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>field_name</i> or <i>field_type</i> is NULL.

See also:

- “`awGet<type>Field`” on page 198
- “`awGet<type>SeqField`” on page 200
- “`awGetField`” on page 241
- “`awGetSequenceField`” on page 248
- “`awGetStructFieldAsEvent`” on page 252
- “`awSet<type>Field`” on page 327
- “`awSet<type>SeqField`” on page 329
- “`awSetField`” on page 345
- “`awSetSequenceField`” on page 349
- “`awSetStructFieldFromEvent`” on page 353

awGetEventId

```
BrokerLong awGetEventId(  
    BrokerEvent event);
```

event The event whose event ID is to be returned.

Returns the event ID, as set by the Broker, for the specified *event*. Returns a value of zero under any one of the following conditions:

- The *event* was created locally and was not received from the Broker.
- The *event* was published by a client using a pre-3.0 release of ActiveWorks.
- The specified *event* is invalid.

awGetEventPublishSequenceNumber

```
BrokerLong awGetEventPublishSequenceNumber(  
    BrokerEvent event);
```

event The event whose publish sequence number is to be returned.

Returns the publish sequence number that the Broker will set for the specified *event* when it is published.

Returns zero if *event* is invalid.

Note:

The `awGetEventPublishSequenceNumber` function can only be used on events that your client application has created and intends to publish. It cannot be used to obtain the `pubSeqn` envelope field from an event received by your client application. See [“Read-only Envelope Fields” on page 34](#).

See also:

“`awPublishEvent`” on page 314

“`awPublishEvents`” on page 315

“`awSetEventPublishSequenceNumber`” on page 344

awGetEventReceiptSequenceNumber

```
BrokerLong awGetEventReceiptSequenceNumber(  
    BrokerEvent event);
```

event The event whose receipt sequence number is to be returned.

Returns the sequence number for the received *event*.

Zero is returned if the event was created locally or if it is a volatile event received from the Broker. Zero may also be returned if *event* is invalid.

See “Using Sequence Numbers” on page 397 for complete information on how to use sequence numbers.

awGetEvents

```
BrokerError awGetEvents(
    BrokerClient client,
    int max_events,
    int msec,
    int *n,
    BrokerEvent **events);
```

<i>client</i>	The Broker client requesting the event.
<i>max_events</i>	The maximum number of events to be returned.
<i>msec</i>	The number of milliseconds to wait for the events before timing out. If set to <code>AW_INFINITE</code> , this function will wait indefinitely.
<i>n</i>	The number of events returned. This parameter is used for output.
<i>events</i>	An array of events. This parameter is used for output.

Acknowledges all previously retrieved events for *client*, then obtains one or more events for *client*, if available. If no events are currently available, this function will wait for the number of milliseconds specified by *msec*. Any *event* that is obtained may be one for which the *client* has registered a subscription, or it may be a delivered event.

If the wait time expires, this function returns a `BrokerError` with a major code of `AW_ERROR_TIMEOUT` and *n* set to zero.

Note:

Before exiting, Broker clients with an explicit-destroy life cycle that are using `awGetEvents` should explicitly acknowledge the receipt sequence number of the last event received using either `awAcknowledge` or `awAcknowledgeThrough`. Failure to do so will result in the last event being received again the next time you connect the Broker client.

Using this function on a client that has registered callback functions will temporarily disable the callback mechanism for that client until this function returns.

The caller is responsible for calling `awDeleteEvent` on each event and then calling `free` on the array itself.

Note:

The Broker will delete all guaranteed events from the event queue once they are acknowledged. If you wish to receive multiple events without acknowledging any previously retrieved events, use the `awGetEventsWithAck` function and specify a sequence number of -1.

Possible BrokerError major codes	Meaning
AW_ERROR_INTERRUPTED	The awInterruptGetEvents function was invoked.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>n</i> or <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>msecs</i> parameter is less than -1 or <i>max_events</i> is less than zero.
AW_ERROR_TIMEOUT	The <i>msecs</i> time-out interval expired before any events arrived.

See also:

“awDispatch” on page 186

“awGetEvent” on page 226

“awGetEventsWithAck” on page 231

“awInterruptGetEvents” on page 279

“awMainLoop” on page 284

“awThreadedCallbacks” on page 371

awGetEventsWithAck

```
BrokerError awGetEventsWithAck(
    BrokerClient client,
    int max_events,
    BrokerLong seqn,
    int msecs,
    int *n,
    BrokerEvent **events);
```

<i>client</i>	The Broker client requesting the events.
<i>max_events</i>	The maximum number of events to be returned.
<i>seqn</i>	Specifies the last event to acknowledge. If set to zero, all previously received events that have not been acknowledged will be acknowledged. If set to -1, no acknowledgment is done at all.
<i>msecs</i>	The number of milliseconds to wait for the events before timing out. If set to AW_INFINITE, this function will wait indefinitely.
<i>n</i>	The number of events returned. This parameter is used for output.
<i>events</i>	An array of events. This parameter is used for output.

Acknowledges all the events received by *client*, up to the event specified by *seqn* and then obtains one or more events. If *seqn* is set to -1, no previously retrieved events will be acknowledged.

The *events* that are obtained may be those for which the *client* has registered a subscription, they may be delivered events, or both.

Calling `awGetEventsWithAck` on a client that has registered callback functions will temporarily disable the callback mechanism for that client until this function returns. For more information on acknowledging events see [“Using Sequence Numbers” on page 397](#). If the wait time expires, this function returns a `BrokerError` with a major code of `AW_ERROR_TIMEOUT` and *n* set to zero.

The caller is responsible for calling `awDeleteEvent` on each event and then for calling `free` on the array itself.

Note:

The Broker will delete all guaranteed events from the event queue once they are acknowledged.

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_INTERRUPTED</code>	The <code>awInterruptGetEvents</code> function was invoked.
<code>AW_ERROR_INVALID_CLIENT</code>	The <i>client</i> has been destroyed or disconnected.
<code>AW_ERROR_NULL_PARAM</code>	The <i>n</i> or <i>events</i> is <code>NULL</code> .
<code>AW_ERROR_OUT_OF_RANGE</code>	The <i>msecs</i> parameter is less than -1 or <i>max_events</i> is less than zero.
<code>AW_ERROR_TIMEOUT</code>	The <i>msecs</i> time-out interval expired before any events arrived.

See also:

“`awDispatch`” on page 186

“`awGetEvent`” on page 226

“`awGetEvents`” on page 230

“`awInterruptGetEvents`” on page 279

“`awMainLoop`” on page 284

“`awThreadedCallbacks`” on page 371

awGetEventTag

```
BrokerError awGetEventTag(
    BrokerEvent event,
    int *tag);
```

event The event whose tag field is to be obtained.

tag The tag field that is obtained. This parameter is used for output.

Obtains the tag envelope field from *event* and places it in *tag*. This is equivalent to, but more convenient than, calling:

```
awGetIntegerField(event, "_env.tag", &tag)
```

Returns an error if the event is invalid, or if the *tag* field is NULL, or if the *_env.tag* field does not exist.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The field <i>_env.tag</i> is not set on the <i>event</i> .
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>tag</i> is NULL.

See also:

“awSetEventTag” on page 345

“awMakeTag” on page 285

awGetEventTypeBaseName

```
char * awGetEventTypeBaseName(
    BrokerEvent event);
```

event The event whose base type is to be returned.

Returns the base name of *event*. The base name does not have the event scope qualification. The caller is responsible for calling *free* on the return value.

A NULL value is returned if *event* is invalid.

See also:

“awGetEventTypeScopeName” on page 239

“awGetEventTypeName” on page 238

awGetEventTypeDef

```
BrokerError awGetEventTypeDef(
    BrokerClient client,
    char *event_type_name,
```

```
BrokerTypeDef *type_def);
```

client The Broker client requesting the event type definition.

event_type_name The name of the event type whose definition is desired.

type_def The event type definition. This parameter is used for output.

Provides the definition of the specified *event_type_name*.

Note:

An event type definition will not be returned if your client is not permitted to browse that event type. In most cases, event types which can be browsed are those which your client can publish or for which it can register subscriptions.

The caller is not responsible for freeing the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish or subscribe to <i>event_type_name</i> .
AW_ERROR_NULL_PARAM	The <i>event_type_def</i> or <i>type_def</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awGetEventTypeDefs” on page 234

“awGetEventTypeName” on page 238

“awGetScopeEventNames” on page 247

“awGetTypeDefFromEvent” on page 273

awGetEventTypeDefs

```
BrokerError awGetEventTypeDefs(
    BrokerClient client,
    int n,
    char **event_type_names,
    BrokerTypeDef **type_defs);
```

client The Broker client requesting the event type definition.

n Number of names in *event_type_names*.

event_type_names An array of event type names.

type_defs An array of event type definitions. This parameter is used for output.

Provides an array of definitions which correspond to the requested event types. For each type in *event_type_names*, a corresponding definition is returned in *defs*. If an event type is undefined or if the *client* does not have permission to obtain the type definition, the corresponding definition is set to NULL.

Note:

An event type definition will not be returned if your client is not permitted to browse that event type. In most cases, event types which can be browsed are those which your client can publish or for which it can register subscriptions.

The caller is responsible for calling `free` on the *defs* array, but not on the definitions themselves.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>event_type_def</i> or <i>type_defs</i> is NULL or one entry in <i>type_defs</i> array is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than or equal to zero.

See also:

“`awGetEventTypeDef`” on page 233

“`awGetEventTypeNames`” on page 238

“`awGetScopeEventTypeNames`” on page 247

“`awGetTypeDefFromEvent`” on page 273

awGetEventTypeFamilyName

```
char * awGetEventTypeFamilyName(BrokerEvent event);
```

This function was replaced by [awGetEventTypeScopeName](#).

awGetEventTypeInfoSet

```
BrokerError awGetEventTypeInfoSet(
    BrokerClient client,
    char *event_type_name,
    char *infoSet_name,
    BrokerEvent *infoSet);
```

client The Broker client requesting the infoSet.

<i>event_type_name</i>	Event type name whose infoSet is to be returned.
<i>infoSet_name</i>	The name of the infoSet.
<i>infoSet</i>	The infoSet that is returned, in event form. This parameter is used for output.

Provides the infoSet for the specified *event_type_name*. The infoSet itself is returned as an event, for convenience. The infoSet name is stored as the event's type name.

The caller is responsible for calling `awDeleteEvent` on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been disconnected or destroyed.
AW_ERROR_NO_PERMISSION	The <i>infoSet_name</i> is not set to "public" and the client does not have permission to publish or subscribe to the event type.
AW_ERROR_NULL_PARAM	The <i>event_type_name</i> , <i>infoSet_name</i> , or <i>infoSet</i> parameter is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The <i>event_type_name</i> does not exist on the Broker.
AW_ERROR_UNKNOWN_INFOSET	The <i>infoSet_name</i> does not exist for the specified event type on the Broker.

See also:

“`awGetEventTypeInfoSetNames`” on page 236

“`awGetEventTypeInfoSets`” on page 237

“`awGetEventTypeNames`” on page 238

“`awGetScopeEventTypeNames`” on page 247

awGetEventTypeInfoSetNames

```
BrokerError awGetEventTypeInfoSetNames(
    BrokerClient client,
    char *event_type_name,
    int *n,
    char ***infoSet_names);
```

<i>client</i>	The Broker client requesting the infoSet names.
<i>event_type_name</i>	The event type name whose infoSets are to be obtained.
<i>n</i>	Number of names in <i>infoSet_names</i> . This parameter is used for output.

infoset_names An array of infoset names. This parameter is used for output.

Provides a list of infosets names for the specified *event_type_name*. The caller is responsible for calling `free` on the output value. The infoset names share one memory block so multiple calls to `free` are not necessary.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been disconnected or destroyed.
AW_ERROR_NO_PERMISSION	The <i>infoset_name</i> is not set to "public" and the client does not have permission to publish or subscribe to the event type.
AW_ERROR_NULL_PARAM	The <i>event_type_name</i> , <i>n</i> , or <i>infoset_names</i> parameter is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The <i>event_type_name</i> does not exist on the Broker.

See also:

“`awGetEventTypeInfoset`” on page 235

“`awGetEventTypeInfosets`” on page 237

“`awGetEventTypeNames`” on page 238

“`awGetScopeEventTypeNames`” on page 247

awGetEventTypeInfosets

```
BrokerError awGetEventTypeInfosets(
    BrokerClient client,
    char *event_type_name,
    int *n,
    char **infoset_names,
    BrokerEvent **infosets);
```

<i>client</i>	The Broker client requesting the infosets.
<i>event_type_name</i>	The event type name whose infoset names are to be obtained.
<i>n</i>	Number of infoset names contained in <i>infoset_names</i> . If set to -1, all infosets will be obtained and this parameter will be changed to the number retrieved. This parameter is used for input and output.
<i>infoset_names</i>	An array containing the names of the infosets to be obtained.
<i>infosets</i>	An array containing the infosets as event type definitions. This parameter is used for output.

Provides the infosets that correspond to the specified infoset names. For each name in *infoset_names*, a corresponding infoset is returned in *infosets*. All infosets are returned as events, for convenience. If an infoset is undefined or not accessible, the corresponding element in *infosets* is set to NULL.

The caller is responsible for calling `awDeleteEvent` on each event in *infosets*, and then calling `free` on the array itself.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been disconnected or destroyed.
AW_ERROR_NULL_PARAM	The parameter <i>event_type_names</i> , <i>n</i> , <i>infoset_names</i> , or <i>infosets</i> is NULL.

See also:

“`awGetEventTypeInfoset`” on page 235

“`awGetEventTypeInfosetNames`” on page 236

“`awGetEventTypeNames`” on page 238

“`awGetScopeEventTypeNames`” on page 247

awGetEventTypeName

```
char * awGetEventTypeName(
    BrokerEvent event);
```

event The event whose name is to be returned.

Gets the fully-qualified name of *event*, which includes the event's base name and scope qualifier. The caller is responsible for calling `free` on the return value.

A NULL value is returned if *event* is invalid.

See also:

“`awGetEventTypeBaseName`” on page 233

“`awGetScopeEventTypeNames`” on page 247

awGetEventTypeNames

```
BrokerError awGetEventTypeNames(
    BrokerClient client,
    int *n,
    char ***event_type_names);
```

<i>client</i>	The Broker client requesting the event type names.
<i>n</i>	Number of event type names returned in <i>names</i> . This parameter is used for output.
<i>event_type_names</i>	The array of event type names. This parameter is used for output.

Provides the list of event type names defined in the Broker to which *client* is connected. The fully qualified names are returned as an array of strings in *event_type_names*.

Note:

Only the names of the event types which your client is permitted to browse are returned. In most cases, this corresponds to the set of event types which your client can publish or for which it can register subscriptions.

The caller is responsible for calling *free* on *event_type_names*. The strings in *event_type_names* share one memory block, so multiple calls to *free* are not necessary.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> or <i>event_type_names</i> is NULL.

See also:

“awGetScopeEventTypeName” on page 247

awGetEventScopeName

```
char * awGetEventScopeName(
    BrokerEvent event);
```

event The event whose scope is to be returned.

Returns the scope of the specified *event*. The scope consists of the qualifier without the base name. The caller is responsible for calling *free* on the return value.

A NULL value is returned if *event* is invalid.

See also:

“awGetEventBaseName” on page 233

“awGetEventName” on page 238

awGetFamilyNames

```
BrokerError awGetFamilyNames(
    BrokerClient client,
    int *n,
    char ***family_names);
```

This has function has been replaced by [awGetScopeNames](#).

awGetFamilyEventTypeNames

```
BrokerError awGetFamilyEventTypeNames(
    BrokerClient client,
    char *family_name,
    int *n,
    char ***event_type_names);
```

This function is replaced by [awGetScopeEventTypeNames](#).

awGetFd

```
BrokerError awGetFd(
    BrokerClient client,
    int *fd);
```

client The Broker client whose field descriptor is to be returned.

fd The client's file descriptor. This parameter is used for output.

Provides the file descriptor for the specified *client*. When data are waiting to be read on the file descriptor, use [awIsClientPending](#) to see if an event has arrived. If data have arrived, you may use any of event retrieval functions, such as [awGetEvent](#), or an event dispatching function, such as [awDispatch](#), to handle the event.

Important:

Use of this function in combination with secure sockets is not recommended.

Each client may have a separate file descriptor or they may share a common descriptor. The sample applications `subscribe4.c` and `subscribe5.c` provide examples of how to use this function.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>fd</i> is NULL.

See also:

“[awIsClientPending](#)” on page 280

“awGetFds” on page 241

“awIsPending” on page 283

awGetFds

```
BrokerError awGetFds(
    int *n,
    int **fds);
```

n The number of file descriptors that were returned. This parameter is used for output.

fds A list of file descriptors for all of this application's Broker clients. This parameter is used for output.

Provides a list of file descriptors for all of this application's Broker clients. When data is waiting to be read on one of the file descriptors, use [awIsClientPending](#) to see if an event has arrived. If data has arrived, you may use any of event retrieval functions, such as [awGetEvent](#), or an event dispatching function, such as [awDispatch](#), to handle the event.

Important:

Use of this function in combination with secure sockets is not recommended.

Possible BrokerError major codes

Meaning

AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> or <i>fds</i> is NULL.

See also:

“awIsClientPending” on page 280

“awGetFd” on page 240

“awIsPending” on page 283

awGetField

```
BrokerError awGetField( BrokerEvent event, char *field_name,
    short *field_type, void **value);
```

event The event containing the desired field.

field_name Name of the desired event field.

field_type The field's type, such as `FIELD_TYPE_BOOLEAN`, defined in `<awetdef.h>`. This parameter is used for output.

value The value of the field. The type of this value depends on the *field_type*. This parameter is used for output.

Provides the value of the field specified by *field_name*. This is a generic access function.

See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

The caller is responsible for freeing the return value in a manner appropriate to the field's type. For a struct type, call `awDeleteEvent` to free the value. To free all other values, simply call `free`.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event or an attempt was made to obtain the value of an envelope field that has not been set.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_INVALID_TYPE	The <i>field_type</i> is FIELD_TYPE_SEQUENCE or is not a supported field type.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> or <i>value</i> is NULL.

See also:

“`awGet<type>Field`” on page 198

“`awGet<type>SeqField`” on page 200

“`awGetEventFieldType`” on page 227

“`awGetSequenceField`” on page 248

“`awGetStructFieldAsEvent`” on page 252

“`awGetStructSeqFieldAsEvents`” on page 253

awGetFieldNames

```
BrokerError awGetFieldNames(
    BrokerEvent event,
    char *field_name,
    int *n,
    char ***field_names);
```

<i>event</i>	The event whose field names are to be returned.
<i>field_name</i>	The field name of a structure containing other fields. If set to NULL, all of the fields in the event are returned.
<i>n</i>	Number of names returned in <i>field_names</i> . This parameter is used for output.
<i>field_names</i>	The array of field names returned. This parameter is used for output.

Provides field names of data fields of the specified *event*. If *field_name* is NULL `awGetFieldNames` gets the fields of the event itself. Otherwise, *field_name* should refer to a field in a structure and this function returns all of the field names at that level.

Note that *field_name* can directly identify any field in the event, no matter how deeply nested the fields may be. See [“Getting Sequence Field Values” on page 29](#) and [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

The caller is responsible for calling `free` on the output value. The strings in *field_names* share one memory block so multiple calls to `free` are not necessary.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> , <i>n</i> , or <i>field_names</i> is NULL.

See also:

“`awGetEventFieldType`” on page 227

awGetFilterEventTypeName

```
char * awGetFilterEventTypeName(
    BrokerFilter filter);
```

filter The filter whose event type name is to be returned.

Returns the *filter* event type name. The caller is responsible for freeing the memory associated with the return value.

This function returns NULL if *filter* is invalid or if memory allocation fails.

See also:

“[awNewBrokerFilter](#)” on page 295

awGetFilterString

```
char * awGetFilterString(  
    BrokerFilter filter);
```

filter The filter whose filter string is to be returned.

Returns the filter string from the specified *filter*.

Returns NULL if *filter* is invalid or if memory allocation fails.

The caller is responsible for freeing the memory associated with the return value. For more information on filters, see “[Managing Event Types](#)” on page 113.

See also:

“[awNewBrokerFilter](#)” on page 295

awGetFloatField

See [awGet<type>Field](#).

awGetFloatSeqField

See [awGet<type>SeqField](#).

awGetIntegerField

See [awGet<type>Field](#).

awGetIntegerSeqField

See [awGet<type>SeqField](#).

awGetLockedClientId

```
BrokerError awGetLockedClientId(  
    ClientQueueLock qlock,  
    char **locked_id);
```

Get information on the locked client as per the client queue lock. Returns the `client_id` of the target client.

Returns `AW_ERROR_INVALID_QUEUE_LOCK` when an invalid `qlock` object is passed. Otherwise returns appropriate information.

awGetLockingClientInfo

```
BrokerError awGetLockingClientInfo(
    ClientQueueLock qlock,
    int *session_id,
    BrokerDate *time_stamp,
    char **locking_id);
```

Get information on the client holding the client queue lock. Returned information includes the client id, `session_id` and the time when the lock was acquired.

Returns `AW_ERROR_INVALID_QUEUE_LOCK` when an invalid `qlock` object is passed. Otherwise returns appropriate information.

awGetLongField

See [awGet<type>Field](#).

awGetLongFieldNative

See [awGet<type>Field](#).

Note:

This function is available only on platforms which support 64-bit integral representations.

See also:

“[awNativetoBrokerLong](#)” on page 290

“[awSetLongFieldNative](#)” on page 348

awGetLongSeqField

See [awGet<type>SeqField](#).

awGetPlatformInfo

```
BrokerError awGetPlatformInfo(
    char *key,
    char **value);
```

key The key of the value to be obtained.

value The value of *key*. This parameter is used for output.

Provides the value of the specified platform *key*.

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAM	The <i>key</i> or <i>value</i> is NULL.
AW_ERROR_UNKNOWN_KEY	The <i>key</i> specified does not exist.

See also:

“awGetPlatformInfoKeys” on page 246

“awSetPlatformInfo” on page 348

awGetPlatformInfoKeys

```
BrokerError awGetPlatformInfoKeys(
    int *n,
    char ***keys);
```

n The number of platform keys in the array. This parameter is used for output.

value An array of platform keys. This parameter is used for output.

Provides a complete list of platform keys. The caller is responsible for freeing the array of strings. The strings are all allocated in a single block, so multiple calls to `free` are not necessary.

The following table shows the keys that are registered by the webMethods Broker library. Other, user-defined keys may also be set using [awSetPlatformInfo](#).

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAM	The parameter <i>n</i> is null.

Key	Value
AdapterLang	"C"
AdapterLangVersion	<current library version>
OS	The name of the operating system under which the caller is executing.
Hardware	The name of the hardware platform on which the caller is executing.

See also:

“awGetPlatformInfo” on page 245

“awSetPlatformInfo” on page 348

awGetScopeEventTypeNameames

```
BrokerError awGetScopeEventTypeNameames(
    BrokerClient client,
    char *scope_name,
    int *n,
    char ***event_type_names);
```

<i>client</i>	The Broker client requesting the event names.
<i>scope_name</i>	The scope of the events to be obtained.
<i>n</i>	The number of strings in <i>names</i> . This parameter is used for output.
<i>event_type_names</i>	An array of strings containing the event type names. This parameter is used for output.

Provides the names of event types for this Broker client in the specified event scope. The names, which are fully qualified, are returned as an array of strings.

Note:

Only the names of the event types which your client is permitted to browse are returned. In most cases, this corresponds to the set of event types which your client can publish or for which it can register subscriptions.

The caller is responsible for calling `free` on *event_type_names*. The strings contained in *event_type_names* share one memory block so multiple calls to `free` are not necessary.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>scope_name</i> , <i>n</i> , or <i>event_type_names</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The <i>scope_name</i> does not exist on the Broker.

See also:

“awGetEventTypeNameames” on page 238

“awGetScopeNames” on page 247

awGetScopeNames

```
BrokerError awGetScopeNames(
    BrokerClient client,
    int *n,
```

```
char ***scope_names);
```

<i>client</i>	The Broker client requesting the scope names.
<i>n</i>	The number of strings in <i>scope_names</i> . This parameter is used for output.
<i>scope_names</i>	An array of strings containing the event scope names. This parameter is used for output.

Provides the event type scope names for the specified client's Broker.

Note:

Only the scope names of the event types which your client is permitted to browse are returned. In most cases, this corresponds to the scope names that contain event types which your client can publish or for which it can register subscriptions.

The caller is responsible for calling *free* on *scope_names*. The strings in *scope_names* share one memory block so multiple calls to *free* are not necessary.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> or <i>scope_names</i> is NULL.

See also:

“awGetEventTypeName” on page 238

“awGetScopeEventTypeName” on page 247

awGetSequenceField

```
BrokerError awGetSequenceField(
    BrokerEvent event,
    char *field_name,
    int offset,
    int max_n,
    short *field_type,
    int *n,
    void **value);
```

<i>event</i>	The event whose sequence field is to be obtained.
<i>field_name</i>	The name of the sequence-type event field.
<i>offset</i>	Number of elements to skip from the beginning of the sequence.
<i>max_n</i>	Number of elements requested this call. If set to -1, all elements will be obtained.

<i>field_type</i>	The type of the fields in the sequence, such as <code>FIELD_TYPE_BOOLEAN</code> , defined in <code><awetdef.h></code> . This parameter is used for output.
<i>n</i>	Number of elements returned. This parameter is used for output.
<i>value</i>	The array of pointers to the values. This parameter is used for output.

Provides the values of the sequence field specified by *field_name*. This is a generic access function. The caller is responsible for appropriately freeing the returned values, based on their type.

If you get a sequence field whose value was not set by the event's publisher, a zero-length sequence will be returned.

See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

For information on how to use *offset*, *max_n*, and *value*; see [awGet<type>SeqField](#).

Possible BrokerError major codes	Meaning
<code>AW_ERROR_FIELD_NOT_FOUND</code>	The <i>field_name</i> does not exist in the event or an attempt was made to obtain the value of an envelope field that has not been set.
<code>AW_ERROR_FIELD_TYPE_MISMATCH</code>	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type.
<code>AW_ERROR_INVALID_EVENT</code>	The <i>event</i> is invalid.
<code>AW_ERROR_INVALID_FIELD_NAME</code>	The <i>field_name</i> is invalid.
<code>AW_ERROR_INVALID_TYPE</code>	The field is a sequence of sequences and cannot be retrieved with this function.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>field_name</i> , <i>n</i> , <i>field_type</i> , or <i>value</i> is NULL.
<code>AW_ERROR_OUT_OF_RANGE</code>	The <i>offset</i> is less than zero or greater than the sequence size, or <i>max_n</i> is less than zero.

See also:

[“awGet<type>Field” on page 198](#)

[“awGet<type>SeqField” on page 200](#)

[“awGetField” on page 241](#)

[“awGetSequenceFieldSize” on page 250](#)

[“awGetStructFieldAsEvent” on page 252](#)

[“awGetStructSeqFieldAsEvents” on page 253](#)

awGetSequenceFieldSize

```
BrokerError awGetSequenceFieldSize(
    BrokerEvent event,
    char *field_name,
    int *size);
```

event The event whose sequence field size is to be obtained.

field_name Name of the sequence field.

size Number of elements in the sequence. This parameter is used for output.

Obtains the number of elements in the sequence specified by *field_name*. See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>field_name</i> does not exist in the event.
AW_ERROR_FIELD_TYPE_MISMATCH	The field is not a sequence or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> or <i>size</i> is NULL.

See also:

“awGet<type>SeqField” on page 200

“awGetSequenceField” on page 248

“awGetStructSeqFieldAsEvents” on page 253

awGetShortField

See [awGet<type>Field](#).

awGetShortSeqField

See [awGet<type>SeqField](#).

awGetSSLCertificate

```
BrokerError awGetSSLCertificate(
    char *certificate_file,
```

```
char *password,
char *trust_file,
BrokerSSLCertificate **certificate);
```

<i>certificate_file</i>	The name of the certificate file.
<i>password</i>	The certificate file's password.
<i>trust_file</i>	The name of the truststore file.
<i>certificate</i>	The certificate from the certificate file. This parameter is used for output.

Obtains the certificate from the *certificate_file*.

The caller is responsible for freeing the *certificate* output value. The `BrokerSSLCertificate` is allocated as a single block of memory, so nested freeing is unnecessary.

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_FILE_NOT_FOUND</code>	The <i>certificate_file</i> could not be found or could not be read.
<code>AW_ERROR_NO_PERMISSION</code>	The <i>password</i> is not valid or the <i>certificate_file</i> does not contain certificates.
<code>AW_ERROR_NULL_PARAM</code>	The <i>certificate_file</i> , <i>password</i> , <i>trust_file</i> , or <i>certificate</i> parameter is NULL.
<code>AW_ERROR_SECURITY</code>	SSL support is not available.

See also:

“`awGetBrokerSSLCertificate`” on page 203

“`awGetSSLEncryptionLevel`” on page 251

awGetSSLEncryptionLevel

```
int awGetSSLEncryptionLevel();
```

Returns one of the following values indicating the SSL encryption level available to the caller.

SSL Level	Description
<code>AW_SSL_LEVEL_NO_ENCRYPTION</code>	Encryption support is not available.
<code>AW_SSL_LEVEL_US_EXPORT</code>	US export level encryption is supported, which limits the size of the keys used.
<code>AW_SSL_LEVEL_US_DOMESTIC</code>	US domestic level encryption is supported, which allows for larger key sizes and greater security.

Note:

US federal regulations restrict the encryption key size in software that is to be exported. Software that is used domestically may use larger encryption key sizes.

See also:

“[awGetClientSSLEncryptionLevel](#)” on page 215

awGetStringField

See [awGet<type>Field](#).

awGetStringSeqField

See [awGet<type>SeqField](#).

awGetStructFieldAsEvent

```
BrokerError awGetStructFieldAsEvent(  
    BrokerEvent event,  
    char *field_name,  
    BrokerEvent *value);
```

<i>event</i>	The event whose field is to be obtained.
<i>field_name</i>	Name of the event field. This may be <code>NULL</code> if you wish to retrieve the top level of the event.
<i>value</i>	An event whose fields correspond to structure fields. This parameter is used for output.

Obtains the specified event field, of type `struct`, and returns it as an event. Each event field corresponds to a field in the structure. Use the appropriate [awGet<type>Field](#) or [awGet<type>SeqField](#) function to obtain the value of the fields within the structure.

Important:

Since the event returned in *value* is not associated with a Broker client, it is not type checked.

If you get a structure field whose value was not set by the event's publisher, a structure will be returned set with all the appropriate default values.

See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

The caller is responsible for calling `awDeleteEvent` on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>field_name</i> does not exist in the event or an attempt was made to obtain the value of an envelope field that has not been set.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.

See also:

“awGet<type>Field” on page 198

“awGet<type>SeqField” on page 200

“awGetField” on page 241

“awGetStructSeqFieldAsEvents” on page 253

“awSetStructFieldFromEvent” on page 353

awGetStructSeqFieldAsEvents

```
BrokerError awGetStructSeqFieldAsEvents(
    BrokerEvent event,
    char *field_name,
    int offset,
    int max_n,
    int *n,
    BrokerEvent **value_array);
```

<i>event</i>	The event containing the sequence field to be obtained.
<i>field_name</i>	The name of an event field that is a sequence of structures.
<i>offset</i>	The number of elements (that is, structures) to skip from the beginning of the sequence.
<i>max_n</i>	The number of elements requested. If set to -1, all elements are requested.
<i>n</i>	The number of elements returned. This parameter is used for output.
<i>value_array</i>	An array of events, each of which represents a structure. This parameter is used for output.

Provides an array of events, each one representing a structure in the sequence contained in *event*. See [awGetStructFieldAsEvent](#) for information on representing a structure as an event. For information on how to use *offset*, *max_n*, and *value_array*; see [awGet<type>SeqField](#).

See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

If you get a structure sequence field whose value was not set by the event's publisher, a zero-length array will be returned.

The caller must call `awDeleteEvent` on each event in *value_array* and then call `free` on *value_array* itself.

Important:

Since the events returned in *value_array* are not associated with a Broker client, they will not type checked.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>field_name</i> does not exist in the event or an attempt was made to obtain the value of an envelope field that has not been set.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> or <i>value_array</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>offset</i> is less than zero or greater than the sequence size, or <i>max_n</i> is less than zero.

See also:

“`awGet<type>Field`” on page 198

“`awGetSequenceField`” on page 248

“`awGetStructFieldAsEvent`” on page 252

“`awSetStructSeqFieldFromEvents`” on page 353

awGetSubscriptionIds

```
int * awGetSubscriptionIds(
    BrokerEvent event,
    int *n);
```

event The event whose subscription identifier is to be checked.

n Number of subscription structures. This parameter is used for output.

Returns the subscription identifiers associated with *event*. A non-NULL return value indicates there are one or more subscriptions that match the specified *event*. In this case, the function returns an array of *n* subscription IDs. See [“BrokerSubscription Objects” on page 67](#) for more information.

If the return value is NULL, the value of *n* indicates the reason:

- -1 indicates a the event was created locally.
- Zero indicates the event was delivered by Broker. Delivered events are not matched to subscriptions.

The caller is responsible for calling `free` on the returned array.

See also:

[“awNewSubscriptionFromStruct” on page 302](#)

[“awNewSubscriptionWithId” on page 305](#)

awGetSubscriptions

```
BrokerError awGetSubscriptions(
    BrokerClient client,
    int *n,
    BrokerSubscription **subs);
```

<i>client</i>	The Broker client whose subscriptions are to be obtained.
<i>n</i>	The number of subscription structures. This parameter is used for output.
<i>subs</i>	An array of subscription structures. This parameter is used for output. See “BrokerSubscription Objects” on page 67 for more information.

Provides an array of subscription structures that represents the subscriptions of the specified *client*.

The caller is responsible for calling `free` on the *subs* output value. All values in *subs* share one memory block so multiple calls to `free` are not necessary.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>n</i> or <i>subs</i> parameter is NULL.

See also:

[“awCancelSubscription” on page 156](#)

[“awCancelSubscriptionFromStruct” on page 157](#)

[“awCancelSubscriptionsFromStructs” on page 157](#)

“awNewSubscription” on page 301

“awNewSubscriptionFromStruct” on page 302

“awNewSubscriptionsFromStructs” on page 303

“awNewSubscriptionWithId” on page 305

awGetTxBrokerSSLCertificate

```
BrokerError awGetTxBrokerSSLCertificate(
    BrokerTxClient txclient,
    BrokerSSLCertificate **certificate);
```

txclient The Broker client whose Broker's SSL certificate is to be returned.

certificate The Broker's certificate. This parameter is used for output.

Obtains the SSL certificate for the Broker to which this *txclient* is connected.

The caller is responsible for freeing the memory associated with the output *certificate*. The BrokerSSLCertificate structure is allocated as one block of memory, so nested freeing is unnecessary.

Possible BrokerError major codes	Meaning
AW_BAD_STATE	The <i>txclient</i> is not using an SSL connection.
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>certificate</i> is NULL.

awGetTxClientAccessLabel

```
BrokerError awGetTxClientAccessLabel(
    BrokerTxClient txclient,
    int *n,
    short **label);
```

txclient The Broker client whose list of event types is to be returned.

n The number of element returned in the *label* array. This parameter is used for output.

label The access label for the client. This parameter is used for output.

Obtains the access label for the specified client. This label will be inserted in the `publabel` envelope field of every event the client publishes or delivers.

Access labels provide access control for event types which is independent of the event type.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	The <i>txclient</i> has an owner, but the access label feature is not enabled.
AW_ERROR_BROKER_FAILURE	The <i>txclient</i> has an owner, but an error occurred looking up the access label.
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have an owner so it cannot have an access label.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> or <i>label</i> is NULL.

See also:

“awSetClientAutomaticControlLabel” on page 332

awGetTxClientApplicationName

```
BrokerError awGetTxClientApplicationName(
    BrokerTxClient txclient,
    char **app_name);
```

<i>txclient</i>	The transactional Broker client whose list of event types is to be returned.
<i>app_name</i>	The name of the application associated with this transactional Broker client. This parameter is used for output.

Provides the application name for this transactional Broker client. The caller is responsible for freeing the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>app_name</i> is NULL.

See also:

“awNewBrokerTxClient” on page 296

“awReconnectBrokerTxClient” on page 321

awGetTxBrokerVersionNumber

```
BrokerError awGetTxBrokerVersionNumber(
    BrokerTxClient txclient,
```

```
int *version);
```

txclient The transactional Broker client whose Broker version number is to be returned.

version The version number, returned as one of the AW_VERSION_* values. This parameter is used for output.

Obtains the version number of the Broker to which this transactional *client* is connected. If the Broker's version number is newer than the client's version, the client's version number is returned.

The caller is responsible for freeing the memory associated with the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>version</i> is NULL.

awGetTxClientBrokerHost

```
BrokerError awGetTxClientBrokerHost(
    BrokerTxClient txclient,
    char **host_name);
```

txclient The transactional Broker client whose Broker host name is to be returned.

host_name The name of the Broker host associated with this Broker client. This parameter is used for output.

Provides a string containing the host name of the Broker for this Broker client. The caller is responsible for calling `free` on the *host_name* string.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> not valid.
AW_ERROR_NULL_PARAM	The parameter <i>host_name</i> is NULL.

See also:

“awGetTxClientBrokerName” on page 259

“awGetTxClientBrokerPort” on page 259

“awNewBrokerTxClient” on page 296

“awReconnectBrokerTxClient” on page 321

awGetTxClientBrokerName

```
BrokerError awGetTxClientBrokerName(
    BrokerTxClient txclient,
    char **broker_name);
```

txclient The transactional Broker client whose Broker name is to be returned.

broker_name The Broker name associated with this transactional Broker client. This parameter is used for output.

Provides a string containing the Broker name for this transactional Broker client. The caller is responsible for calling `free` on the *broker_name* string.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>broker_name</i> is NULL.

See also:

“awGetTxClientBrokerHost” on page 258

“awGetTxClientBrokerPort” on page 259

“awNewBrokerTxClient” on page 296

“awReconnectBrokerTxClient” on page 321

awGetTxClientBrokerPort

```
BrokerError awGetTxClientBrokerPort(
    BrokerTxClient txclient,
    int *port_number);
```

txclient The transactional Broker client whose client Broker port is to be returned.

port_number The port number of the Broker Server. This parameter is used for output.

Provides a int value containing the port number of the transactional client's Broker Server.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>port_number</i> is NULL.

See also:

“awGetTxClientBrokerHost” on page 258

“awGetTxClientBrokerName” on page 259

“awNewBrokerTxClient” on page 296

“awReconnectBrokerTxClient” on page 321

awGetTxClientConnectionDescriptor

```
BrokerError awGetTxClientConnectionDescriptor(
    BrokerTxClient txclient,
    BrokerConnectionDescriptor *desc);
```

<i>txclient</i>	The transactional Broker client whose connection descriptor is to be returned.
<i>desc</i>	The transactional Broker client's connection descriptor. This parameter is used for output.

Obtains the connection descriptor for the transactional client and returns it in *desc*. If a NULL descriptor argument was passed to `awNewBrokerClient` when *client* was created, then a descriptor which matches the default connection descriptor behavior is retrieved.

Note:

This function cannot be used to get the actual Broker settings for a connection descriptor.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>desc</i> is NULL.

awGetTxClientGroup

```
BrokerError awGetTxClientGroup(
    BrokerTxClient txclient,
    char **group_name);
```

<i>txclient</i>	The Broker client whose transactional client group name is to be returned.
<i>group_name</i>	The name of the transactional client group to which this Broker client belongs. This parameter is used for output.

Creates a string containing the transactional client's group name. The caller is responsible for calling `free` on the *group_name* string. For more information on client groups, see *Administering webMethods Broker*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>group_name</i> is NULL.

See also:

“awNewBrokerTxClient” on page 296

“awReconnectBrokerTxClient” on page 321

awGetTxClientId

```
BrokerError awGetTxClientId(
    BrokerTxClient txclient,
    char **client_id);
```

txclient The Broker client whose identifier is to be returned.

client_id The transactional client id that is associated with this Broker client. This parameter is used for output.

Creates a string containing the transactional client's identifier. The caller is responsible for calling free on the *client_id* string.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>client_id</i> is NULL.

See also:

“awNewBrokerTxClient” on page 296

“awReconnectBrokerTxClient” on page 321

awGetTxClientInfoSet

```
BrokerError awGetTxClientInfoSet(
    BrokerTxClient txclient,
    BrokerEvent *infoSet);
```

txclient The Broker client whose infoSet is to be returned.

infoSet The area where the retrieved infoSet is returned. This parameter is used for output.

Obtains the infoSet for the specified *txclient*. The infoSet is returned as a Broker event for convenience. Each client can store one infoSet that can be used to contain information about its state or configuration.

The caller is responsible for invoking [awDeleteEvent](#) on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>infoSet</i> is NULL.

See also:

“awSetClientInfoSet” on page 332

awGetTxClientQueueLength

```
BrokerError awGetTxClientQueueLength(
    BrokerTxClient txclient,
    int *n);
```

<i>txclient</i>	The Broker client whose queue length is to be returned.
<i>n</i>	The number of events currently in the Broker client's event queue. This parameter is used for output.

Provides the number of events currently in event queue for *txclient*.

Note:

The length returned will include any unacknowledged events in the queue. This means the length may be greater than you expect if your Broker client has received events, but has not yet acknowledged them.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> is NULL.

See also:

“awTxCanPublish” on page 373

awGetTxClientLastPublishSequenceNumber

```
BrokerError awGetTxClientLastPublishSequenceNumber(
    BrokerTxClient txclient,
```

```
BrokerLong *seqn);
```

txclient The Broker client whose last transactionally published sequence number is to be returned.

seqn The sequence number last published by the Broker client. This parameter is used for output.

Obtains the highest sequence number used by the *txclient* for publishing or delivering an event.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>txclient</i> parameter, has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>seqn</i> parameter is NULL.

awGetTxClientSSLBrokerDistinguishedName

```
BrokerError awGetTxClientSSLBrokerDistinguishedName(
    BrokerTxClient txclient,
    char **distinguished_name);
```

Get the secure socket distinguished name for the Broker. The caller is responsible for freeing the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	The <i>txclient</i> is not using a secure socket connection.
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> is NULL.

awGetTxClientSSLBrokerIssuerDistinguishedName

```
BrokerError awGetTxClientSSLBrokerIssuerDistinguishedName(
    BrokerTxClient txclient,
    char **distinguished_name);
```

Get the secure socket distinguished name of the issuer of the Broker's certificate. The caller is responsible for freeing the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	The <i>txclient</i> is not using a secure socket connection.
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAM	The parameter <i>n</i> is NULL.

awGetTxClientSSLEncryptionLevel

```
BrokerError awGetTxClientSSLEncryptionLevel(
    BrokerTxClient txclient,
    int *level);
```

<i>txclient</i>	The Broker client whose SSL encryption level is to be returned.
<i>level</i>	The SSL level being used. See the following table for a list of possible values. This parameter is used for output.

Returns an indication of the level of SSL that is supported as one of the values shown below. If SSL is not being used by this client, a `AW_BAD_STATE` exception is thrown.

SSL Level	Description
AW_SSL_LEVEL_NO_ENCRYPTION	Encryption support is not available.
AW_SSL_LEVEL_US_EXPORT	US export level encryption is supported, which limits the size of the keys used.
AW_SSL_LEVEL_US_DOMESTIC	US domestic level encryption is supported, which allows for larger key sizes and greater security.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	The <i>txclient</i> is not using SSL.
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>level</i> is NULL.

See also:

“awGetSSLEncryptionLevel” on page 251

awGetTxClientStateShareLimit

```
BrokerError awGetTxClientStateShareLimit(
    BrokerTxClient txclient,
    int *limit);
```

<i>txclient</i>	The Broker client whose state share limit is to be returned.
-----------------	--------------------------------------------------------------

limit The maximum number of clients that may share this Broker client's state. This parameter is used for output.

Provides the maximum number of Broker clients that can share *txclient*'s state. The transactional client state includes the Broker client's event queue and list of subscribed events. If the limit returned is -1, this indicates there is no limit to the number of Broker clients that can share this state.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>limit</i> is NULL.

See also:

“awSetCharField” on page 332

“awSetDescriptorSSLEncrypted” on page 342

awGetTxClientTerritoryName

```
BrokerError awGetTxClientTerritoryName(
    BrokerTxClient txclient,
    char **territory_name);
```

txclient The Broker client whose Broker's territory name is to be returned.

territory_name The area where the name is to be returned. This parameter is used for output.

Returns the territory name associated with the Broker to which the transactional client is connected. The caller is responsible for calling `free` on the returned value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> is not valid.
AW_ERROR_NULL_PARAM	The parameter <i>territory_name</i> is NULL.

See also:

“awGetTypeDefTerritoryName” on page 274

awGetTxDefaultBrokerPort

```
int awGetTxDefaultBrokerPort();
```

Returns the default number this transactional client uses for connecting to Brokers. The default port is used for non-SSL connections and has a value of 6849. The default port number is also used to calculate the port numbers shown in [“Basic Properties” on page 50](#).

See also:

“awSetDefaultClientTimeout” on page 336

awGetTxExternalId

```
BrokerError awGetTxExternalId(  
    BrokerTxClient txclient,  
    char **ext_id);
```

<i>txclient</i>	The Broker client whose external identifier is to be returned.
<i>ext_id</i>	Used by application or external manager to identify the context. Set when the context is created. It is constant for the life of the context. This id is optional. It does not have to be unique and can be null. Returns an external identifier.

awGetTxId

```
BrokerError awGetTxId(  
    BrokerTxClient txclient,  
    BrokerLong *tx_id);
```

<i>txclient</i>	The Broker client whose state is to be returned.
<i>tx_id</i>	The identifier of the transaction, created with the awMakeTxTransactionId function.

Returns the unique Id set by the Broker for this transaction. Also, any of the communications errors can occur.

awGetTxState

```
BrokerError awGetTxState(  
    BrokerTxClient txclient,  
    int *tx_state);
```

<i>txclient</i>	The Broker client whose state is to be returned.
<i>tx_state</i>	

Gets the transaction's state. Caller is responsible for freeing the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The client is not valid.
AW_ERROR_NULL_PARAM	The host_name is null.

awGetTxSubscriptions

```
BrokerError awGetTxSubscriptions(
    BrokerTxClient txclient,
    int *n,
    BrokerSubscription **subs);
```

<i>txclient</i>	The Broker client whose subscriptions are to be obtained.
<i>n</i>	The number of subscription structures. This parameter is used for output.
<i>subs</i>	An array of subscription structures. This parameter is used for output. See “BrokerSubscription Objects” on page 67 for more information.

Provides an array of subscription structures that represents the subscriptions of the specified *txclient*.

The caller is responsible for calling `free` on the *subs* output value. All values in *subs* share one memory block so multiple calls to `free` are not necessary.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>n</i> or <i>subs</i> parameter is NULL.

See also:

- “[awCancelTxSubscription](#)” on page 158
- “[awCancelTxSubscriptionFromStruct](#)” on page 159
- “[awCancelTxSubscriptionsFromStructs](#)” on page 160
- “[awNewTxSubscription](#)” on page 306
- “[awNewTxSubscriptionFromStruct](#)” on page 307
- “[awNewTxSubscriptionsFromStructs](#)” on page 308
- “[awNewTxSubscriptionWithId](#)” on page 309

awGetTypeDefBaseName

```
char * awGetTypeDefBaseName(
    BrokerTypeDef type_def);
```

type_def The event type definition whose name is to be returned.

Returns the name of the event type definition, without the scope qualifier, from the event type definition. Returns NULL if the *type_def* is not valid.

The caller is responsible for calling `free` on the output value.

See also:

“awGetTypeDefFieldDef” on page 270

“awGetTypeDefStorageType” on page 274

awGetTypeDefBrokerHost

```
BrokerError awGetTypeDefBrokerHost(
    BrokerTypeDef type_def,
    char **host_name);
```

type_def The event type definition.

host_name The name of the Broker host. This parameter is used for output.

Obtains the host name of the Broker host where this event type is defined.

The caller is responsible for calling `free` on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_TYPEDEF	The parameter <i>type_def</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>host_name</i> is NULL.

See also:

“awGetTypeDefBrokerName” on page 268

“awGetTypeDefBrokerPort” on page 269

awGetTypeDefBrokerName

```
BrokerError awGetTypeDefBrokerName(
    BrokerTypeDef type_def,
```

```
char **broker_name);
```

type_def The event type definition.

broker_name The name of Broker Server. This parameter is used for output.

Obtains the name of the Broker where this event type is defined.

The caller is responsible for calling `free` on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_TYPEDEF	The parameter <i>type_def</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>broker_name</i> is NULL.

See also:

“awGetTypeDefBrokerHost” on page 268

“awGetTypeDefBrokerPort” on page 269

awGetTypeDefBrokerPort

```
BrokerError awGetTypeDefBrokerPort(
    BrokerTypeDef type_def,
    int *port_number);
```

type_def The event type definition.

port_number The port number of Broker Server. This parameter is used for output.

Obtains the port number of the Broker Server where this event type is defined.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_TYPEDEF	The parameter <i>type_def</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>port_number</i> is NULL.

See also:

“awGetTypeDefBrokerHost” on page 268

“awGetTypeDefBrokerName” on page 268

awGetTypeDefDescription

```
BrokerError awGetTypeDefDescription(
    BrokerTypeDef type_def,
    char **description);
```

<i>type_def</i>	The type definition whose scope is to be returned.
<i>description</i>	The description associated with the type definition. This parameter is used for output.

Returns the description associated with the event type specified by *type_def*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_TYPE	The parameter <i>type_def</i> does not specify an event type.
AW_ERROR_INVALID_TYPEDEF	The parameter <i>type_def</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>description</i> is NULL.

awGetTypeDefFamilyName

```
char * awGetTypeDefFamilyName(BrokerTypeDef type_def);
```

This function has been replaced by [awGetTypeDefScopeName](#).

awGetTypeDefFieldDef

```
BrokerError awGetTypeDefFieldDef(
    BrokerTypeDef type_def,
    char *field_name,
    BrokerTypeDef *field_def);
```

<i>type_def</i>	The event type definition.
<i>field_name</i>	The name of a field whose type definition is to be obtained.
<i>field_def</i>	The field's type definition. This parameter is used for output.

Obtains the field definition for the field with *field_name*. See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>field_name</i> does not exist in the event type.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_TYPE_MISMATCH	The <i>field_name</i> incorrectly accesses a type, such as using a subscript on a non-sequence field.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_INVALID_TYPEDEF	The <i>type_def</i> parameter is invalid.
AW_ERROR_NULL_PARAM	The <i>field_name</i> or <i>field_def</i> parameter is NULL.

See also:

“awGetTypeDefFieldType” on page 272

“awGetTypeDefFieldNames” on page 271

awGetTypeDefFieldNames

```
BrokerError awGetTypeDefFieldNames(
    BrokerTypeDef type_def,
    char *field_name,
    int *n,
    char ***names);
```

<i>type_def</i>	The event type definition.
<i>field_name</i>	The name of a structure field whose sub-fields are to be obtained. If set to NULL, the fields at the top level of the event are obtained.
<i>n</i>	The number of field names returned in <i>names</i> . This parameter is used for output.
<i>names</i>	An array of event type definitions. This parameter is used for output.

Obtains an array of strings that represents the field names of the specified event type or field. In addition, *n* is set to the number of strings returned in *names*.

See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

The caller is responsible for calling `free` on *names*. All of the strings in *names* share one memory block so multiple calls to free are not necessary.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>field_name</i> does not exist in the event type.
AW_ERROR_FIELD_TYPE_MISMATCH	The <i>field_name</i> incorrectly accesses a type, such as using a subscript on a non-sequence field.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_TYPEDEF	The <i>type_def</i> parameter is invalid.
AW_ERROR_NULL_PARAM	The <i>field_name</i> , <i>n</i> , or <i>names</i> parameter is NULL.

awGetTypeDefFieldType

```
BrokerError awGetTypeDefFieldType(
    BrokerTypeDef type_def,
    char *field_name,
    short *field_type);
```

<i>type_def</i>	The event type definition.
<i>field_name</i>	The field name whose type is to be obtained.
<i>field_type</i>	A type value. This parameter is used for output.

Obtains the data type of the specified field, as specified by the defined constants in `awetdef.h`:

```
FIELD_TYPE_BYTE
FIELD_TYPE_SHORT
FIELD_TYPE_INT
FIELD_TYPE_LONG
FIELD_TYPE_FLOAT
FIELD_TYPE_DOUBLE
FIELD_TYPE_BOOLEAN
FIELD_TYPE_DATE
FIELD_TYPE_CHAR
FIELD_TYPE_SEQUENCE
FIELD_TYPE_STRING
FIELD_TYPE_STRUCT
FIELD_TYPE_UNICODE_CHAR
FIELD_TYPE_UNICODE_STRING
FIELD_TYPE_UNKNOWN
```

See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>field_name</i> does not exist in the event type.
AW_ERROR_FIELD_TYPE_MISMATCH	The <i>field_name</i> incorrectly accesses a type, such as using a subscript on a non-sequence field.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_INVALID_TYPEDEF	The <i>type_def</i> parameter is invalid.
AW_ERROR_NULL_PARAM	The <i>field_name</i> or <i>field_type</i> parameter is NULL.

See also:

“awGetTypeDefFieldDef” on page 270

awGetTypeDefFromEvent

```
BrokerError awGetTypeDefFromEvent(
    BrokerEvent event,
    BrokerTypeDef *type_def);
```

event The event whose type definition is to be retrieved.

type_def The type definition. This parameter is used for output.

Obtains the event type definition for the specified *event*.

The caller is not responsible for freeing the return value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_TYPEDEF	The <i>event</i> is not associated with a Broker client.
AW_ERROR_NULL_PARAM	The parameter <i>type_def</i> is NULL.

See also:

“awGetCanPublishTypeDefs” on page 205

“awGetCanSubscribeTypeDefs” on page 207

“awGetEventTypeDef” on page 233

“awGetEventTypeDefs” on page 234

awGetTypeDefScopeName

```
char * awGetTypeDefScopeName(
    BrokerTypeDef type_def);
```

type_def The type definition whose scope is to be returned.

Returns the scope of the event type specified by *type_def*.

The caller is responsible for calling `free` on the return value.

A NULL value is returned if *type_def* is invalid.

See also:

“awGetTypeDefBaseName” on page 268

“awGetTypeDefStorageType” on page 274

awGetTypeDefStorageType

```
BrokerError awGetTypeDefStorageType(
    BrokerTypeDef type_def,
    int *storage_type);
```

type_def The type definition whose name is to be returned.

storage_type The location where the storage type value is to be returned. This parameter is used for output. Returns a value of AW_STORAGE_GUARANTEED.

Obtains the storage type for this event type.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_TYPEDEF	The <i>event</i> is not associated with a Broker client.
AW_ERROR_NULL_PARAM	The parameter <i>time_to_live</i> is NULL.

awGetTypeDefTerritoryName

```
BrokerError awGetTypeDefTerritoryName(
    BrokerTypeDef type_def,
    char **territory_name);
```

type_def The type definition whose Broker's territory name is to be returned.

territory_name The area where the name is to be returned. This parameter is used for output.

Returns the territory name of the Broker where the specified event type definition is defined. The caller is responsible for calling free on the returned value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>territory_name</i> is NULL.

See also:

“awGetClientTerritoryName” on page 216

awGetTypeDefTimeToLive

```
BrokerError awGetTypeDefTimeToLive(
    BrokerTypeDef type_def,
    int *time_to_live);
```

type_def The type definition whose name is to be returned.

time_to_live The location where the time-to-live value is to be returned. This parameter is used for output.

Obtains the time, in seconds, that an event type will be available after being published before it expires or is destroyed. A return value of zero indicates the event will never expire.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_TYPEDEF	The <i>event</i> is not associated with a Broker client.
AW_ERROR_NULL_PARAM	The parameter <i>time_to_live</i> is NULL.

awGetTypeDefTypeName

```
char * awGetTypeDefTypeName(
    BrokerTypeDef type_def);
```

type_def The type definition whose name is to be returned.

Returns the fully qualified name of the specified Broker event type, *type_def*. The fully qualified name consists of the base name with its scope qualifier. NULL is returned if *type_def* is invalid.

The caller is responsible for calling `free` on the return value.

See also:

“awGetTypeDefBaseName” on page 268

“awGetTypeDefFieldDef” on page 270

awGetUCCharField

See [awGet<type>Field](#).

awGetUCCharSeqField

See [awGet<type>SeqField](#).

awGetUCStringField

See [awGet<type>Field](#).

awGetUCStringFieldAsA

```
BrokerError awGetUCStringFieldAsA(
    BrokerEvent event,
    char *field_name,
    char **value);
```

event The event from which the string is to be retrieved.

field_name The name of the unicode string field to be retrieved.

value The ANSI string that was retrieved. This parameter is used for output.

Obtains an ANSI string field with the specified unicode string *field_name* from the *event*. The string returned in *value* will be in ANSI format, which encodes 16-bit Unicode characters using an escape notation, such as `\u1234`.

The caller is not responsible for freeing the return value.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>field_name</i> does not exist in the event type or an attempt was made to obtain the value of an envelope field that has not been set.
AW_ERROR_FIELD_TYPE_MISMATCH	The <i>field_name</i> incorrectly accesses a type, such as using a subscript on a non-sequence field.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The <i>field_name</i> or <i>value</i> parameter is NULL.

See also:

“[awGet<type>Field](#)” on page 198

“[awGet<type>SeqField](#)” on page 200

“[awGetUCStringFieldAsUTF8](#)” on page 277

“[awGet<type>SeqField](#)” on page 200

“[awGetStructFieldAsEvent](#)” on page 252

“[awGetStructSeqFieldAsEvents](#)” on page 253

awGetUCStringFieldAsUTF8

```
BrokerError awGetUCStringFieldAsUTF8(
    BrokerEvent event,
    char *field_name,
    charUC **value);
```

event The event from which the string is to be retrieved.

field_name The name of the string field to be retrieved.

value The string that was retrieved. This parameter is used for output.

Obtains a Unicode string field with the specified *field_name* from the specified *event*. The string returned in *value* will be in Unicode Transform Function 8 (UTF-8) format, which encodes 16-bit Unicode characters in multi-byte, 8-bit encoding. This means that the string returned in *value* may contain as many as four logical characters for each Unicode character.

The caller is not responsible for freeing the return value.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>field_name</i> does not exist in the event type or an attempt was made to obtain the value of an envelope field that has not been set.
AW_ERROR_FIELD_TYPE_MISMATCH	The <i>field_name</i> incorrectly accesses a type, such as using a subscript on a non-sequence field.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The <i>field_name</i> or <i>value</i> parameter is NULL.

See also:

“[awGetUCStringSeqFieldAsUTF8](#)” on page 278

“[awSetUCStringFieldAsUTF8](#)” on page 358

awGetUCStringSeqField

See [awGet<type>SeqField](#).

awGetUCStringSeqFieldAsUTF8

```
BrokerError awGetUCStringSeqFieldAsUTF8(
    BrokerEvent event,
    char *field_name,
    int offset,
    int max_n,
    int *n,
    charUC ***value);
```

<i>event</i>	The event from which the string sequence is to be retrieved.
<i>field_name</i>	The name of the sequence field to be retrieved.
<i>offset</i>	The number of elements to skip from the beginning of the sequence.
<i>max_n</i>	The number of elements requested. If set to -1, all elements are requested.
<i>n</i>	The number of elements actually returned. This parameter is used for output.
<i>value</i>	An array of values associated with the field. This parameter is used for output.

Obtains a Unicode string sequence field with the specified *field_name* from the specified *event*. The array of values returned will be in Unicode Transform Function 8 (UTF-8) format, which encodes 16-bit Unicode characters in multi-byte, 8-bit encoding. This means that each string returned may contain as many as four logical characters for each Unicode character.

See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

For information on how to use *offset*, *max_n*, and *value*; see [awGet<type>SeqField](#).

The caller is not responsible for freeing the return value.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_TYPEDEF	The <i>event</i> is not associated with a Broker client.
AW_ERROR_NULL_PARAM	The parameter <i>type_def</i> is NULL.

See also:

“awGetUCStringFieldAsUTF8” on page 277

“awSetUCStringSeqFieldAsUTF8” on page 360

awInit

awInitForThreads

```
BrokerError awInitForThreads();
```

This function should be called by a multi-threaded client, prior to making any other API function calls.

awInterrupt

awInterruptDispatch

```
BrokerError awInterruptDispatch();
```

Interrupts the current `awDispatch` call. If there is no `awDispatch` in progress, the next call to `awDispatch` will be interrupted. The `awInterruptDispatch` function is intended for use in multi-threaded client applications and is safe for use in signal handlers.

See also:

“`awDispatch`” on page 186

awInterruptGetEvents

```
BrokerError awInterruptGetEvents(
    BrokerClient client);
```

client The client whose get events call is to be interrupted.

Interrupts the current `awGetEvent` or `awGetEvents` call for the specified *client*. If there is no such call in progress, the next call to `awGetEvent` or `awGetEvents` will be interrupted. This function is intended for use in multi-threaded clients and is safe for use in a signal handler.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.

See also:

“`awGetEvent`” on page 226

“`awGetEvents`” on page 230

“`awGetEventsWithAck`” on page 231

awls

awlsAckReplyEvent

```
BrokerBoolean awIsAckReplyEvent(  
    BrokerEvent event);
```

event The event to be tested.

Determine if *event* is an acknowledgment reply to an earlier request event. Returns true (1) if the event is of type `Adapter::ack`. Returns false(0) if the above is not true, or if the event is invalid.

See also:

“awIsErrorReplyEvent” on page 281

“awIsLastReplyEvent” on page 282

“awIsNullReplyEvent” on page 283

awlsClientConnected

```
BrokerBoolean awIsClientConnected(  
    BrokerClient client);
```

client The Broker client whose connection is to be tested.

Returns 0 (false) if the specified *client* was disconnected by a call to either `awDisconnectClient`, or `awDestroyClient`, or by an error; otherwise, 1 (true) is returned.

Important:

This function does a passive check on the connection-it does not access the Broker client over the network. Therefore, this function may erroneously report that a client is connected when it actually is not connected.

awlsClientPending

```
BrokerError awIsClientPending(  
    BrokerClient client,  
    BrokerBoolean *is_pending);
```

client The Broker client to be checked.

is_pending Set to 1 (true) if requests are pending; otherwise, set to 0 (false) . This parameter is used for output.

Checks if any events are pending for the specified *client*. Events will be pending only if a previous call to [awPrimeClient](#) or [awPrimeAllClients](#) has been made.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>is_pending</i> is NULL.

See also:

“awGetFd” on page 240

“awGetFds” on page 241

“awIsPending” on page 283

“awPrimeAllClients” on page 313

“awPrimeClient” on page 314

awIsClientQueueLocked

```
BrokerError awIsClientQueueLocked(ClientQueueLock qllock, BrokerBoolean
*is_locked);
```

Checks whether the qllock object holds a client queue lock on a target client. Returns true if the target is locked, otherwise false.

Returns AW_ERROR_INVALID_QUEUE_LOCK when an invalid qllock object is passed. Otherwise returns appropriate information.

awIsErrorReplyEvent

```
BrokerBoolean awIsErrorReplyEvent(
BrokerEvent event);
```

event The event to be tested.

Determine if *event* is an error reply to an earlier request event. Returns true (1) if the event is of type `Adapter::error`. Returns false(0) if the above is not true, or if the event is invalid.

See also:

“awIsAckReplyEvent” on page 280

“awIsLastReplyEvent” on page 282

“awIsNullReplyEvent” on page 283

awIsEventFieldSet

```
BrokerError awIsEventFieldSet(
    BrokerEvent event,
    char *field_name,
    BrokerBoolean *is_set);
```

<i>event</i>	The event whose field is to be checked.
<i>field_name</i>	The field name to check.
<i>is_set</i>	Set to 1 (true) if the field is set; otherwise, set to 0 (false) . This parameter is used for output.

Determines if the specified field was explicitly set to its current value. The *field_name* can directly identify any field in an event, no matter how deeply the fields are nested. See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

See [“Sequence Data Fields” on page 27](#) and [awGetFieldNames](#) for more information.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
AW_ERROR_FIELD_TYPE_MISMATCH	The <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> or <i>is_set</i> is NULL.

awIsLastReplyEvent

```
BrokerBoolean awIsLastReplyEvent(
    BrokerEvent event);
```

<i>event</i>	The event to be tested.
--------------	-------------------------

Returns true (1) if the event is an `Adapter::error` event or if this event is the last in a reply sequence, indicated by the envelope fields `appSeqn` and `appLastSeqn` being equal. Otherwise 0 (false) is returned.

See also:

“awIsAckReplyEvent” on page 280

“awIsErrorReplyEvent” on page 281

“awIsLastReplyEvent” on page 282

awIsNullReplyEvent

```
BrokerBoolean awIsNullReplyEvent(
    BrokerEvent event);
```

event The event to be tested.

Returns 1 (true) if this event is a null reply, indicated by the envelope fields appSeqn and appLastSeqn both being equal to -1. Otherwise 0 (false) is returned.

See also:

“awIsAckReplyEvent” on page 280

“awIsLastReplyEvent” on page 282

“awIsNullReplyEvent” on page 283

awIsPending

```
BrokerError awIsPending(BrokerBoolean *is_pending);
```

is_pending Set to 1 (true) if events are pending; otherwise, set to 0 (false). This parameter is used for output.

Checks if any events are pending for any client of the application. The `subscribe4.c` sample application provides an example that uses this function.

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAM	The parameter <i>is_pending</i> is NULL.

See also:

“awGetFd” on page 240

“awGetFds” on page 241

“awIsClientPending” on page 280

“awPrimeAllClients” on page 313

“awPrimeClient” on page 314

awIsTxClientConnected

```
BrokerBoolean awIsTxClientConnected(  
    BrokerTxClient txclient);
```

txclient The transactional Broker client whose connection is to be tested.

Returns 0 (*false*) if the specified *txclient* was disconnected by a call to either `awDisconnectTxClient`, `awDestroyTxClient`, or by an error; otherwise, 1 (*true*) is returned.

Important:

This function does a passive check on the connection-it does not access the transactional Broker client over the network. Therefore, this function may erroneously report that a client is connected when it actually is not connected.

awLock

awLockTypeDefCache

```
void awLockTypeDefCache();
```

Locks the application's type cache, thereby prevents any flush operations. See [“Event Type Definition Cache” on page 117](#) for more information on using this function.

See also:

“`awFlushTypeDefCache`” on page 198

“`awUnlockTypeDefCache`” on page 394

awMain

awMainLoop

```
BrokerError awMainLoop();
```

Starts a main-loop process that executes until stopped. The `awMainLoop` function is logically equivalent to:

```
while(do_not_stop) { awDispatch(AW_INFINITE); }
```

When using `awMainLoop`, you cannot invoke any of the following functions:

- `awDispatch`
- `awThreadedCallbacks`

For more information on the callback model see [“Using Specific Callbacks” on page 80](#).

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_BAD_STATE	This function was called from a callback function.

See also:

- “awDispatch” on page 186
- “awGetEvent” on page 226
- “awGetEvents” on page 230
- “awGetEventsWithAck” on page 231
- “awStopMainLoop” on page 362
- “awThreadedCallbacks” on page 371

awMake

awMakeSubId

```
int awMakeSubId(
    BrokerClient client);
```

client The client for whom the subscription ID is being made.

Returns a subscription ID for the specified *client*.

Note:

It is possible for this function to create duplicate subscription IDs for subscriptions that were opened before this Broker client session was created. You should use this function only with clients that do not make use of `awDisconnectClient` and `awReconnectClient`. See [awMakeUniqueSubId](#) for a function that creates subscription IDs that are unique across multiple sessions.

See also:

- “awMakeUniqueSubId” on page 288
- “awNewSubscriptionFromStruct” on page 302
- “awNewSubscriptionsFromStructs” on page 303
- “awNewSubscriptionWithId” on page 305

awMakeTag

```
int awMakeTag(BrokerClient client);
```

client The Broker client for whom the tag is being created.

Returns a unique value for use in the `tag` envelope field of an event. The `tag` field is used in the request-reply model to associate a reply event with its corresponding request event. Returns 0 if the client has been disconnected or destroyed.

See also:

“`awGetEvents`” on page 230

“`awSetEventTag`” on page 345

awMakeTransactionId

```
BrokerError awMakeTransactionId(
    BrokerClient client,
    char **transaction_id);
```

client The Broker client for whom the transaction ID is to be created.

transaction_id The generated identifier that is returned. This parameter is used for output.

Generates a unique transaction identifier for use in transaction operations.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>transaction_id</i> is NULL.

See also:

“`awBeginTransaction`” on page 150

“`awEndTransaction`” on page 188

awMakeTxSubId

```
int awMakeTxSubId(
    BrokerTxClient txclient);
```

txclient The *txclient* for whom the subscription ID is being made.

Returns a subscription ID for the specified *txclient*.

Note:

It is possible for this function to create duplicate subscription IDs for subscriptions that were opened before this Broker client session was created. You should use this function only with transactional clients that do not make use of `awDisconnectClient` and `awReconnectClient`. See [awMakeUniqueSubId](#) for a function that creates subscription IDs that are unique across multiple sessions.

See also:

“`awMakeUniqueSubId`” on page 288

“`awNewTxSubscriptionFromStruct`” on page 307

“`awNewTxSubscriptionsFromStructs`” on page 308

“`awNewTxSubscriptionWithId`” on page 309

awMakeTxTag

```
int awMakeTxTag(
    BrokerTxClient txclient);
```

txclient The Broker client for whom the tag is being created.

Returns a unique value for use in the `tag` envelope field of an event. The tag field is used in the request-reply model to associate a reply event with its corresponding request event. Returns 0 if the transactional client has been disconnected or destroyed.

See also:

“`awGetEvents`” on page 230

“`awSetEventTag`” on page 345

awMakeTxTransactionId

```
BrokerError awMakeTxTransactionId(
    BrokerTxClient txclient,
    char **transaction_id);
```

txclient The Broker client for whom the transaction ID is to be created.

transaction_id The generated identifier that is returned. This parameter is used for output.

Generates a unique transaction identifier for use in transaction operations. Transaction IDs are formatted as follows: “`clientId:UniqueNumber:CountNumber`”.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>transaction_id</i> is NULL.

See also:

“awTxBeginTransaction” on page 372

“awEndTransaction” on page 188

awMakeUniqueSubId

```
BrokerError awMakeUniqueSubId(
    BrokerClient client,
    int *sub_id);
```

client The Broker client for whom the subscription ID is to be created.

sub_id The subscription ID. This parameter is used for output.

Creates a subscription ID for the specified *client* and returns it in the output parameter *sub_id*. This function queries the Broker for the client's open subscriptions and excludes any subscription IDs that are already in use. Use this function for clients that may have subscriptions that are open from a previous session.

After initially calling `awMakeUniqueSubId`, subsequent calls to `awMakeSubId` will return unique subscription IDs.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>sub_id</i> parameter is NULL.

See also:

“awMakeSubId” on page 285

“awNewSubscriptionFromStruct” on page 302

“awNewSubscriptionsFromStructs” on page 303

“awNewSubscriptionWithId” on page 305

awMakeUniqueTxSubId

```
BrokerError awMakeUniqueTxSubId(
```

```
BrokerTxClient txclient,
int *sub_id);
```

txclient The Broker client for whom the subscription ID is to be created.

sub_id The subscription ID. This parameter is used for output.

Creates a subscription ID for the specified *txclient* and returns it in the output parameter *sub_id*. This function queries the Broker for the client's open subscriptions and excludes any subscription IDs that are already in use. Use this function for clients that may have subscriptions that are open from a previous session.

After initially calling `awMakeUniqueTxSubId`, subsequent calls to `awMakeTxSubId` will return unique subscription IDs.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>sub_id</i> parameter is NULL.

See also:

“`awMakeTxSubId`” on page 286

“`awNewTxSubscriptionFromStruct`” on page 307

“`awNewTxSubscriptionsFromStructs`” on page 308

“`awNewTxSubscriptionWithId`” on page 309

awMalloc

awMalloc

```
void *awMalloc(
size_t size);
```

size The number of bytes being allocated.

Returns a pointer to a block of memory of the specified *size*. This function returns a NULL pointer if memory could not be allocated.

Memory will be allocated from the same heap as other functions provided in the webMethods Broker API, which allows you to use the `awFree` function consistently throughout your application.

See also:

“`awFree`” on page 197

“awRealloc” on page 319

awMatch

awMatchFilter

```
BrokerError awMatchFilter(
    BrokerFilter filter,
    BrokerEvent event,
    BrokerBoolean *result);
```

<i>filter</i>	The filter to use to check the event.
<i>event</i>	The event to be checked.
<i>result</i>	Set to 1 (true) if <i>filter</i> matches <i>event</i> , otherwise set to 0 (false). This parameter is used for output.

Determines if the specified *filter* matches the specified *event*.

Possible BrokerError major codes	Meaning
AW_ERROR_FILTER_RUNTIME	An error occurred while evaluating the <i>filter</i> .
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FILTER	The <i>filter</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>result</i> is NULL.

See also:

“awNewBrokerFilter” on page 295

awNative

awNativetoBrokerLong

```
NativeLong awNativetoBrokerLong(
    BrokerLong bl);
```

Returns a NativeLong containing the number represented in the specified BrokerLong.

Note:

This function is available only on platforms which support 64-bit integral representations.

See also:

“awBrokerToNativeLong” on page 152

“awGetBrokerLong” on page 203

“awSetBrokerLong” on page 331

awNew

awNewBrokerClient

```
BrokerError awNewBrokerClient(
    char *broker_host,
    char *broker_name,
    char *client_id,
    char *client_group,
    char *app_name,
    BrokerConnectionDescriptor desc,
    BrokerClient *client);
```

<i>broker_host</i>	The name of the host running the Broker that the new Broker client will use. Specified in the form <broker_host_name>:<port_number>. If you omit the port number, the default port number will be used. For example: MyHost:12000
<i>broker_name</i>	The name of the Broker to which the new Broker client will be connected. If NULL, the default Broker will be used. If specified, the length of this parameter must be 1 to 255 ANSI characters or 1 to 42 Unicode characters.
<i>client_id</i>	A string containing a unique identifier for the new Broker client to be used when disconnecting or reconnecting. If NULL, the Broker will generate a unique client identifier. A NULL value is usually supplied for clients whose life cycle is <i>destroy-on-disconnect</i> . See “ Client Identifiers ” on page 45 for more information.
<i>client_group</i>	The name of the client group for the new Broker client. Client groups are discussed in <i>Administering webMethods Broker</i> .
<i>app_name</i>	The name of the application that will contain the new Broker client.
<i>desc</i>	The connection descriptor to use for the new Broker client. If NULL, a default connection will be established for the new client, which may be shared with other Broker clients.
<i>client</i>	A pointer to the newly created BrokerClient. This parameter is used for output.

Creates a client. *broker_name* can be NULL to request the default Broker. *client_id* can be NULL to request the Broker to create an identifier (usually used with *destroy-on-disconnect* clients). *desc* can be NULL to create a default connection.

The caller is responsible for calling *awDestroyClient* on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_BROKER_NOT_RUNNING	A Broker could not be found on the specified <i>broker_host</i> .
AW_ERROR_CLIENT_EXISTS	Another Broker client is using the specified <i>client_id</i> .
AW_ERROR_COMM_FAILURE	A problem occurred during connection establishment.
AW_ERROR_HOST_NOT_FOUND	The <i>broker_host</i> could not be found.
AW_ERROR_INVALID_CLIENT_ID	The <i>client_id</i> contains invalid characters.
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.
AW_ERROR_NO_PERMISSION	Permission to join the <i>client_group</i> was denied.
AW_ERROR_NULL_PARAM	The <i>broker_host</i> , <i>client_group</i> , <i>app_name</i> , or <i>client</i> parameter is NULL.
AW_ERROR_SECURITY	A secure connection was attempted, but was rejected by the Broker.
AW_ERROR_UNKNOWN_BROKER_NAME	The specified <i>broker_name</i> does not exist. If <i>broker_name</i> is NULL, this indicates that there is no default Broker.
AW_ERROR_UNKNOWN_CLIENT_GROUP	The specified <i>client_group</i> does not exist on the Broker.
AW_ERROR_BASIC_AUTH_FAILURE	Basic authentication failed for the user.
AW_ERROR_INVALID_BASICAUTH_VALUE	You did not specify a password for the basic authentication user.

See also:

“awDisconnectClient” on page 184

“awDestroyClient” on page 183

“awNewOrReconnectBrokerClient” on page 298

“awReconnectBrokerClient” on page 319

awNewBrokerConnectionDescriptor

```
BrokerConnectionDescriptor
awNewBrokerConnectionDescriptor();
```

Creates a new Broker descriptor with connection sharing enabled and state sharing disabled.

By default, the descriptor will have a null SSL certificate file will be set to true. For more information on SSL, see [“Managing Event Types” on page 113](#).

Returns NULL if memory could not be allocated.

You are responsible for calling [awDeleteDescriptor](#) on the returned value.

See also:

“awNewBrokerClient” on page 291

“awNewOrReconnectBrokerClient” on page 298

“awReconnectBrokerClient” on page 319

awNewBrokerDate

```
BrokerDate * awNewBrokerDate(
    int yr,
    int mo,
    int dy,
    int hr,
    int m,
    int s,
    int ms);
```

<i>yr</i>	The four-digit year, such as 1996.
<i>mo</i>	The one or two-digit month.
<i>dy</i>	The one or two-digit day.
<i>hr</i>	The hour in 24 hour format (0 through 23)
<i>m</i>	The minute (0 through 59)
<i>s</i>	The second.
<i>ms</i>	The millisecond (0 through 999).

Creates a Broker date, initializes it with the date and time parameters, and sets `is_date_and_time` to 1 (`true`). The caller is responsible for calling [awDeleteDate](#) on the returned date.

Returns `NULL` if memory could not be allocated.

See also:

“awCopyDate” on page 165

“awDeleteDate” on page 167

“awNewBrokerDateSequence” on page 294

“awNewEmptyBrokerDate” on page 297

awNewBrokerDateSequence

```
BrokerDate * awNewBrokerDateSequence(
    int n);
```

n The number of dates to be created.

Creates an array with *n* empty Broker dates. The `is_date_and_time` field is set to 0 (false). The caller is responsible for calling `awDeleteDate` on the return value.

Returns NULL if memory could not be allocated.

See also:

“awCopyDate” on page 165

“awDeleteDate” on page 167

“awNewBrokerDate” on page 293

“awNewEmptyBrokerDate” on page 297

awNewBrokerEvent

```
BrokerError awNewBrokerEvent(
    BrokerClient client,
    char *event_type_name,
    BrokerEvent *event);
```

client The Broker client for which the new event is to be created. If set to NULL, the event is not type checked.

event_type_name The event type name.

event The newly created event. This parameter is used for output.

Creates the *event* with the type specified by *event_type_name* for the specified *client*. The caller is responsible for calling `awDeleteEvent` on the output value.

If *client* is not NULL, this function will check the names and types of the event's fields, one by one, as they are set. If you set a field with a type that does not match the event type definition stored in the Broker, an error will be returned. This notifies you that you need to update your application to keep it synchronized with the latest event type definitions managed by the Broker.

For example, if your application adds a field named `xx` by calling `awSetIntegerField` and the field is defined to be of type `int`, an error will be returned.

Note:

If *client* is set to NULL, no type checking will occur when this event's fields are retrieved or set. However, the event's type name will be recorded in the event for later use.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	A non-NULL <i>client</i> parameter was specified and it is invalid.
AW_ERROR_INVALID_EVENT_TYPE_NAME	The <i>event_type_name</i> is invalid.
AW_ERROR_NO_PERMISSION	A non-NULL <i>client</i> parameter was specified and that client does not have permission to publish or subscribe to the event type.
AW_ERROR_NULL_PARAM	The parameter <i>event_type_name</i> or <i>event</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	A non-NULL <i>client</i> parameter was specified and the event type was not found on the Broker.

See also:

“awCopyEvent” on page 166

“awDeleteEvent” on page 168

awNewBrokerFilter

```
BrokerError awNewBrokerFilter(
    BrokerClient client,
    char *event_type_name,
    char *filter_string,
    BrokerFilter *filter);
```

<i>client</i>	The Broker client for which this filter is being created.
<i>event_type_name</i>	The name of the event type to be filtered. A wildcard may be added to the end of the event type name. See “Using Wildcards” on page 62 for more information.
<i>filter_string</i>	The filter string.
<i>filter</i>	The newly created filter. This parameter is used for output.

Creates *filter*, using the specified *filter_string*. The caller is responsible for calling `awDeleteFilter` on the return value.

Possible BrokerError major codes	Meaning
AW_ERROR_FILTER_PARSE	An error occurred while attempting to parse <i>filter_string</i> .
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>event_type_name</i> is NULL.

See also:

“[awDeleteFilter](#)” on page 169

“[awMatchFilter](#)” on page 290

awNewBrokerString

```
BrokerString awNewBrokerString();
```

Creates a `BrokerString` that may be used as a field in a `BrokerEvent`. The caller must use [awDeleteString](#) to free the return value.

Returns `NULL` if memory could not be allocated.

See also:

“[awDeleteString](#)” on page 169

“[awDeleteStringWrapper](#)” on page 169

awNewBrokerTxClient

```
BrokerError awNewBrokerTxClient(  
    char broker_host,  
    char *broker_name,  
    char *client_id,  
    char *client_group,  
    char *app_name,  
    BrokerConnectionDescriptor desc,  
    BrokerTxClient txclient);
```

<i>broker_host</i>	The name of the host which is running the Broker that the transactional Broker client will use. Specified in the form <code><broker_host_name>:<port_number></code> . If you omit the port number, the default port number will be used. For example: <code>MyHost:12000</code>
<i>broker_name</i>	The name of the Broker to which the transactional Broker client will be connected. If <code>NULL</code> , the default Broker will be used. If specified, the length of this parameter must be 1 to 255 ANSI characters or 1 to 42 Unicode characters.
<i>client_id</i>	A string containing a unique identifier for creating or reconnecting the transactional Broker client. See “ Client Identifiers ” on page 45 for more information. The client id can be <code>NULL</code> to request the Broker to create an identifier (usually used with destroy-on-disconnect clients).
<i>client_group</i>	The name of the client group for the transactional Broker client. Client groups are discussed in <i>Administering webMethods Broker</i> .

<i>app_name</i>	The name of the application that will contain the transactional Broker client.
<i>desc</i>	The connection descriptor to use for the transactional Broker client. If NULL, a default connection will be established for the new client, which may be shared with other Broker clients.
<i>txclient</i>	A pointer to the reconnected or newly created transactional Broker client. This parameter is used for output.

Attempts to create a new transactional Broker client with an event queue and zero subscriptions. If a Broker client with the same *client_id* already exists, this client will be reconnected. After invoking this function, the Broker client may begin processing events.

The client state sharing property of *desc* will be ignored if the Broker client already exists and is being reconnected

The caller is responsible for calling `awDestroyClient` on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_BROKER_NOT_RUNNING	The host exists but no Broker is running on that host.
AW_ERROR_CLIENT_EXISTS	The specified <i>client_id</i> is already in use by another Broker client.
AW_ERROR_COMM_FAILURE	A problem occurred during connection establishment.
AW_ERROR_HOST_NOT_FOUND	The <i>broker_host</i> could not be found.
AW_ERROR_INVALID_CLIENT_ID	The <i>client_id</i> contains invalid characters.
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.
AW_ERROR_INVALID_NAME	The <i>app_name</i> parameter is invalid.
AW_ERROR_NO_PERMISSION	Permission to join the specified <i>client_group</i> was denied.
AW_ERROR_NULL_PARAM	The <i>broker_host</i> , <i>client_group</i> , <i>app_name</i> , or <i>client</i> parameter is NULL.
AW_ERROR_SECURITY	A secure connection was attempted, but was rejected by the Broker.
AW_ERROR_UNKNOWN_BROKER_NAME	The specified <i>broker_name</i> does not exist. If <i>broker_name</i> is NULL, this indicates that there is no default Broker.
AW_ERROR_UNKNOWN_CLIENT_GROUP	The specified client group does not exist on the Broker.

awNewEmptyBrokerDate

```
BrokerDate * awNewEmptyBrokerDate();
```

Creates an empty date suitable for use as a field in a `BrokerEvent`, setting the `is_date_and_time` field to 0 (`false`). The caller is responsible for calling `awDeleteDate` on the return value.

Returns `NULL` if memory could not be allocated.

See also:

“`awCopyDate`” on page 165

“`awDeleteDate`” on page 167

“`awNewBrokerDate`” on page 293

“`awNewBrokerDateSequence`” on page 294

awNewOrReconnectBrokerClient

```
BrokerError awNewOrReconnectBrokerClient(
    char *broker_host,
    char *broker_name,
    char *client_id,
    char *client_group,
    char *app_name,
    BrokerConnectionDescriptor desc,
    BrokerClient *client);
```

<i>broker_host</i>	The name of the host where the Broker is executing that the Broker client will use. Specified in the form <code><broker_host_name>:<port_number></code> . If you omit the port number, the default port number will be used. For example: <code>MyHost:12000</code>
<i>broker_name</i>	The name of the Broker to which the Broker client will be connected. If <code>NULL</code> , the default Broker will be used. If specified, the length of this parameter must be 1 to 255 ANSI characters or 1 to 42 Unicode characters.
<i>client_id</i>	A string containing a unique identifier for creating or reconnecting the Broker client. See “ Client Identifiers ” on page 45 for more information.
<i>client_group</i>	The name of the client group for the Broker client. Client groups are discussed in <i>Administering webMethods Broker</i> .
<i>app_name</i>	The name of the application that will contain the Broker client.
<i>desc</i>	The connection descriptor to use for the Broker client. If <code>NULL</code> , a default connection will be established for the new client, which may be shared with other Broker clients.
<i>client</i>	A pointer to the reconnected or newly created Broker client. This parameter is used for output.

Attempts to create a new Broker client with an event queue and zero subscriptions. If a Broker client with the same *client_id* already exists, this client will be reconnected. After invoking this function, the Broker client may begin processing events.

The client state sharing property of *desc* will be ignored if the Broker client already exists and is being reconnected.

The application is responsible for invoking either the `awDisconnectClient` or `awDestroyClient` function when the Broker client is no longer needed.

Possible BrokerError major codes	Meaning
AW_ERROR_BROKER_NOT_RUNNING	The host exists but no Broker is running on that host.
AW_ERROR_CLIENT_CONTENTION	The specified <i>client_id</i> is already in use by another Broker client. If state sharing is enabled, this indicates the maximum number of Broker clients has been exceeded.
AW_ERROR_COMM_FAILURE	A problem occurred during connection establishment.
AW_ERROR_HOST_NOT_FOUND	The <i>broker_host</i> could not be found.
AW_ERROR_INVALID_CLIENT_ID	The <i>client_id</i> contains invalid characters.
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.
AW_ERROR_INVALID_NAME	The <i>app_name</i> parameter is invalid.
AW_ERROR_NO_PERMISSION	Permission to join the specified <i>client_group</i> was denied.
AW_ERROR_NULL_PARAM	The <i>broker_host</i> , <i>client_id</i> , <i>client_group</i> , <i>app_name</i> , or <i>client</i> parameter is NULL.
AW_ERROR_SECURITY	A secure connection was attempted, but was rejected by the Broker.
AW_ERROR_UNKNOWN_BROKER_NAME	The specified <i>broker_name</i> does not exist. If <i>broker_name</i> is NULL, this indicates that there is no default Broker.
AW_ERROR_UNKNOWN_CLIENT_GROUP	if the specified client group does not exist on the Broker.
AW_ERROR_UNKNOWN_CLIENT_ID	The specified <i>client_id</i> does not exist on the Broker.
AW_ERROR_BASIC_AUTH_FAILURE	Basic authentication failed for the user.
AW_ERROR_INVALID_BASICAUTH_VALUE	You did not specify a password for the basic authentication user.

See also:

“`awDisconnectClient`” on page 184

“`awDestroyClient`” on page 183

“awNewBrokerClient” on page 291

“awReconnectBrokerClient” on page 319

awNewOrReconnectBrokerTxClient

```
BrokerError awNewOrReconnectBrokerTxClient(
    char *broker_host,
    char *broker_name,
    char *client_id,
    char *client_group,
    char *app_name,
    BrokerConnectionDescriptor desc,
    BrokerTxClient *txclient);
```

<i>broker_host</i>	The name of the host which is running the Broker that the transactional Broker client will use. Specified in the form <broker_host_name>:<port_number>. If you omit the port number, the default port number will be used. For example: MyHost:12000
<i>broker_name</i>	The name of the Broker to which the Broker client will be connected. If NULL, the default Broker will be used. If specified, the length of this parameter must be 1 to 255 ANSI characters or 1 to 42 Unicode characters.
<i>client_id</i>	A string containing a unique identifier for creating or reconnecting the Broker client. See “ Client Identifiers ” on page 45 for more information.
<i>client_group</i>	The name of the client group for the Broker client. Client groups are discussed in <i>Administering webMethods Broker</i> .
<i>app_name</i>	The name of the application that will contain the Broker client.
<i>desc</i>	The connection descriptor to use for the Broker client. If NULL, a default connection will be established for the new client, which may be shared with other Broker clients.
<i>txclient</i>	A pointer to the reconnected or newly created Broker client. This parameter is used for output.

Attempts to create a new transactional Broker client with an event queue and zero subscriptions. If a Broker client with the same *client_id* already exists, this client will be reconnected. After invoking this function, the Broker client may begin processing events.

The client state sharing property of *desc* will be ignored if the Broker client already exists and is being reconnected.

The application is responsible for invoking either the `awDisconnectClient` or `awDestroyClient` function when the Broker client is no longer needed.

Possible BrokerError major codes	Meaning
AW_ERROR_BROKER_NOT_RUNNING	The host exists but no Broker is running on that host.

Possible BrokerError major codes	Meaning
AW_ERROR_CLIENT_CONTENTION	The specified <i>client_id</i> is already in use by another Broker client. If state sharing is enabled, this indicates the maximum number of Broker clients has been exceeded.
AW_ERROR_COMM_FAILURE	A problem occurred during connection establishment.
AW_ERROR_HOST_NOT_FOUND	The <i>broker_host</i> could not be found.
AW_ERROR_INVALID_CLIENT_ID	The <i>client_id</i> contains invalid characters.
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.
AW_ERROR_INVALID_NAME	The <i>app_name</i> parameter is invalid.
AW_ERROR_NO_PERMISSION	Permission to join the specified <i>client_group</i> was denied.
AW_ERROR_NULL_PARAM	The <i>broker_host</i> , <i>client_id</i> , <i>client_group</i> , <i>app_name</i> , or <i>client</i> parameter is NULL.
AW_ERROR_SECURITY	A secure connection was attempted, but was rejected by the Broker.
AW_ERROR_UNKNOWN_BROKER_NAME	The specified <i>broker_name</i> does not exist. If <i>broker_name</i> is NULL, this indicates that there is no default Broker.
AW_ERROR_UNKNOWN_CLIENT_GROUP	if the specified client group does not exist on the Broker.
AW_ERROR_UNKNOWN_CLIENT_ID	The specified <i>client_id</i> does not exist on the Broker.

awNewSubscription

```
BrokerError awNewSubscription(
    BrokerClient client,
    char *event_type_name,
    char *filter);
```

<i>client</i>	The Broker client requesting the subscription.
<i>event_type_name</i>	The event type name for the subscription.
<i>filter</i>	The event filter. Set to NULL if event filtering is not desired.

Subscribes the specified *client* to events of the specified event type that match the filter string. If *filter* is NULL, all events of the specified event type will be considered to match the subscription.

An asterisk can be added to the end of an event type name as a wildcard if you want to open a subscription for multiple event types.

Note:

When using wildcards in the *event_type_name*, the *filter* may contain envelope fields but it may not contain any data fields. Wildcards cannot be used with the `Broker::Trace` or `Broker::Activity` event types.

If the subscription has already been registered, calling this function will have no effect.

This function implicitly passes a `subId` field of zero; see [awNewSubscriptionWithId](#).

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	The <i>filter</i> string contains a parse error.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to subscribe to the event type.
AW_ERROR_NULL_PARAM	The <i>event_type_name</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“[awCanSubscribe](#)” on page 153

“[awCancelSubscription](#)” on page 156

“[awCancelSubscriptionFromStruct](#)” on page 157

“[awCancelSubscriptionsFromStructs](#)” on page 157

“[awDoesSubscriptionExist](#)” on page 187

“[awGetSubscriptions](#)” on page 255

“[awNewSubscriptionFromStruct](#)” on page 302

“[awNewSubscriptionsFromStructs](#)” on page 303

“[awNewSubscriptionWithId](#)” on page 305

awNewSubscriptionFromStruct

```
BrokerError awNewSubscriptionFromStruct(
    BrokerClient client,
    BrokerSubscription *sub);
```

client The Broker client requesting the subscription.

sub The subscription structure. See “[BrokerSubscription Objects](#)” on page 67 for more information.

Subscribes the specified *client* to the events specified by the subscription structure *sub*. The subscription structure should be initialized with the desired subscription identifier, event name, and filter string.

If the subscription has already been registered, calling this function will have no effect. A subscription with a *sub_id* of 1 will not be replaced with a new subscription with a *sub_id* of 2 that otherwise is identical.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	The filter string contains a parse error.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to subscribe to the event type.
AW_ERROR_NULL_PARAM	The <i>sub</i> parameter or the event type name it contains is NULL.
AW_ERROR_OUT_OF_RANGE	The subscription identifier contained in <i>sub</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

- “awCanSubscribe” on page 153
- “awCancelSubscription” on page 156
- “awCancelSubscriptionFromStruct” on page 157
- “awCancelSubscriptionsFromStructs” on page 157
- “awDoesSubscriptionExist” on page 187
- “awGetSubscriptions” on page 255
- “awNewSubscription” on page 301
- “awNewSubscriptionsFromStructs” on page 303
- “awNewSubscriptionWithId” on page 305

awNewSubscriptionsFromStructs

```
BrokerError awNewSubscriptionsFromStructs(
    BrokerClient client,
    int n,
    BrokerSubscription *subs);
```

<i>client</i>	The Broker client requesting the subscriptions.
<i>n</i>	The number of subscription structures in the array.
<i>subs</i>	An array of subscription structures. See “BrokerSubscription Objects” on page 67 for more information.

Subscribes the specified *client* to the events specified by the subscription structures in the array *subs*. Each of the subscription structures should be initialized with the desired subscription identifier, event name, and filter string.

If one or more of the subscriptions in *subs* has already been registered, calling this function will have no effect on those subscriptions. A subscription with a *sub_id* of 1 will not be replaced with a new subscription with a *sub_id* of 2 that otherwise is identical.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	One of the <i>subs</i> filter strings contains a parse error.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to subscribe to one of the <i>subs</i> event types.
AW_ERROR_NULL_PARAM	The parameter <i>sub</i> , or one of the subscriptions it contains, is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>sub_id</i> parameter or one of the subscriptions is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type of one of the subscriptions does not exist on the Broker.

See also:

- “awCancelSubscription” on page 156
- “awCancelSubscriptionFromStruct” on page 157
- “awCancelSubscriptionsFromStructs” on page 157
- “awDoesSubscriptionExist” on page 187
- “awGetSubscriptions” on page 255
- “awNewSubscription” on page 301
- “awNewSubscriptionFromStruct” on page 302
- “awNewSubscriptionWithId” on page 305

awNewSubscriptionWithId

```
BrokerError awNewSubscriptionWithId(
    BrokerClient client,
    int sub_id,
    char *event_type_name,
    char *filter);
```

<i>client</i>	The Broker client requesting the subscription.
<i>sub_id</i>	The subscription ID.
<i>event_type_name</i>	The event type name for the subscription.
<i>filter</i>	The event filter string. Set to NULL if event filtering is not desired.

Subscribes the specified *client* to events of the specified event type that match the filter string. If *filter* is NULL, all events of the given event type are considered to match the subscription. The *sub_id* will be associated with any event received by client that matches the *event_type_name* and *filter*. An asterisk can be added to the end of an event type name as a wildcard if you want to open a subscription for multiple event types.

Note:

When using wildcards in the *event_type_name*, the *filter* may contain envelope fields but it may not contain any data fields. Wildcards cannot be used with the `Broker::Trace` or `Broker::Activity` event types.

If the subscription has already been registered, calling this function will have no effect. A subscription with a *sub_id* of 1 will not be replaced with a new subscription with a *sub_id* of 2 that otherwise is identical. Subscription IDs do not uniquely identify a particular subscription, so you can create different subscriptions and specify the same subscription ID. See [“Uniqueness of Subscriptions” on page 61](#) for more information on what differentiates one subscription from another.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	The <i>filter</i> string contains a parse error.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to subscribe to the event type.
AW_ERROR_NULL_PARAM	The <i>event_type_name</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>sub_id</i> parameter is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awCanSubscribe” on page 153

“awCancelSubscription” on page 156

“awCancelSubscriptionFromStruct” on page 157

“awCancelSubscriptionsFromStructs” on page 157

“awDoesSubscriptionExist” on page 187

“awGetSubscriptions” on page 255

“awNewSubscription” on page 301

“awNewSubscriptionFromStruct” on page 302

“awNewSubscriptionsFromStructs” on page 303

awNewTxSubscription

```
BrokerError awNewTxSubscription(
    BrokerTxClient txclient,
    char *event_type_name,
    char *filter);
```

<i>txclient</i>	The Broker client requesting the subscription.
<i>event_type_name</i>	The event type name for the subscription.
<i>filter</i>	The event filter. Set to NULL if event filtering is not desired.

Subscribes the specified *txclient* to events of the specified event type that match the filter string. If *filter* is NULL, all events of the specified event type will be considered to match the subscription.

An asterisk can be added to the end of an event type name as a wildcard if you want to open a subscription for multiple event types.

Note:

When using wildcards in the *event_type_name*, the *filter* may contain envelope fields but it may not contain any data fields. Wildcards cannot be used with the `Broker::Trace` or `Broker::Activity` event types.

If the subscription has already been registered, calling this function will have no effect.

This function implicitly passes a `subId` field of zero; see [awNewSubscriptionWithId](#).

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	The <i>filter</i> string contains a parse error.

Possible BrokerError major codes	Meaning
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to subscribe to the event type.
AW_ERROR_NULL_PARAM	The <i>event_type_name</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awCancelSubscription” on page 156

“awCancelSubscriptionFromStruct” on page 157

“awCancelSubscriptionsFromStructs” on page 157

“awDoesSubscriptionExist” on page 187

“awGetTxSubscriptions” on page 267

“awNewSubscriptionFromStruct” on page 302

“awNewSubscriptionsFromStructs” on page 303

“awNewSubscriptionWithId” on page 305

“awTxCanSubscribe” on page 373

awNewTxSubscriptionFromStruct

```
BrokerError awNewTxSubscriptionFromStruct(
    BrokerTxClient txclient,
    BrokerSubscription *sub);
```

txclient The Broker client requesting the subscription.

sub The subscription structure. See “[BrokerSubscription Objects](#)” on page 67 for more information.

Subscribes the specified *txclient* to the events specified by the subscription structure *sub*. The subscription structure should be initialized with the desired subscription identifier, event name, and filter string.

If the subscription has already been registered, calling this function will have no effect. A subscription with a *sub_id* of 1 will not be replaced with a new subscription with a *sub_id* of 2 that otherwise is identical.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_SUBSCRIPTION	The filter string contains a parse error.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to subscribe to the event type.
AW_ERROR_NULL_PARAM	The <i>sub</i> parameter or the event type name it contains is NULL.
AW_ERROR_OUT_OF_RANGE	The subscription identifier contained in <i>sub</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

- “awCanSubscribe” on page 153
- “awCancelSubscription” on page 156
- “awCancelSubscriptionFromStruct” on page 157
- “awCancelSubscriptionsFromStructs” on page 157
- “awDoesSubscriptionExist” on page 187
- “awGetTxSubscriptions” on page 267
- “awNewTxSubscription” on page 306
- “awNewSubscriptionsFromStructs” on page 303
- “awNewSubscriptionWithId” on page 305

awNewTxSubscriptionsFromStructs

```
BrokerError awNewTxSubscriptionsFromStructs(
    BrokerTxClient txclient,
    int n,
    BrokerSubscription *subs);
```

- txclient* The Broker client requesting the subscriptions.
- n* The number of subscription structures in the array.
- subs* An array of subscription structures. See “[BrokerSubscription Objects](#)” on [page 67](#) for more information.

Subscribes the specified *txclient* to the events specified by the subscription structures in the array *subs*. Each of the subscription structures should be initialized with the desired subscription identifier, event name, and filter string.

If one or more of the subscriptions in *subs* has already been registered, calling this function will have no effect on those subscriptions. A subscription with a *sub_id* of 1 will not be replaced with a new subscription with a *sub_id* of 2 that otherwise is identical.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	One of the <i>subs</i> filter strings contains a parse error.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to subscribe to one of the <i>subs</i> event types.
AW_ERROR_NULL_PARAM	The parameter <i>sub</i> , or one of the subscriptions it contains, is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>sub_id</i> parameter or one of the subscriptions is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type of one of the subscriptions does not exist on the Broker.

See also:

- “awCancelSubscription” on page 156
- “awCancelSubscriptionFromStruct” on page 157
- “awCancelSubscriptionsFromStructs” on page 157
- “awDoesSubscriptionExist” on page 187
- “awGetTxSubscriptions” on page 267
- “awNewTxSubscription” on page 306
- “awNewTxSubscriptionFromStruct” on page 307
- “awNewTxSubscriptionWithId” on page 309

awNewTxSubscriptionWithId

```
BrokerError awNewTxSubscriptionWithId(
    BrokerTxClient txclient,
    int sub_id,
    char *event_type_name,
    char *filter);
```

- txclient* The Broker client requesting the subscription.
- sub_id* The subscription ID.

<i>event_type_name</i>	The event type name for the subscription.
<i>filter</i>	The event filter string. Set to NULL if event filtering is not desired.

Subscribes the specified *txclient* to events of the specified event type that match the filter string. If *filter* is NULL, all events of the given event type are considered to match the subscription. The *sub_id* will be associated with any event received by client that matches the *event_type_name* and *filter*. An asterisk can be added to the end of an event type name as a wildcard if you want to open a subscription for multiple event types.

Note:

When using wildcards in the *event_type_name*, the *filter* may contain envelope fields but it may not contain any data fields. Wildcards cannot be used with the `Broker::Trace` or `Broker::Activity` event types.

If the subscription has already been registered, calling this function will have no effect. A subscription with a *sub_id* of 1 will not be replaced with a new subscription with a *sub_id* of 2 that otherwise is identical. Subscription IDs do not uniquely identify a particular subscription, so you can create different subscriptions and specify the same subscription ID. See [“Uniqueness of Subscriptions” on page 61](#) for more information on what differentiates one subscription from another.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_INVALID_SUBSCRIPTION	The <i>filter</i> string contains a parse error.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to subscribe to the event type.
AW_ERROR_NULL_PARAM	The <i>event_type_name</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>sub_id</i> parameter is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

- “awCancelSubscription” on page 156
- “awCancelSubscriptionFromStruct” on page 157
- “awCancelSubscriptionsFromStructs” on page 157
- “awDoesSubscriptionExist” on page 187
- “awGetTxSubscriptions” on page 267
- “awNewTxSubscription” on page 306
- “awNewTxSubscriptionFromStruct” on page 307

“awNewSubscriptionsFromStructs” on page 303

“awTxCanSubscribe” on page 373

awParse

awParseLongFromString

```
BrokerError awParseLongFromString(
    char *start,
    char **next,
    char *terminator,
    BrokerLong *output);
```

<i>start</i>	The string to be parsed.
<i>next</i>	The character immediately following the character that caused parsing to terminate. This parameter is used for output.
<i>terminator</i>	The character that caused parsing to terminate. This parameter is used for output.
<i>output</i>	The extracted value. This parameter is used for output.

Extracts a `BrokerLong` type from the string specified by *start*. Parsing will proceed until a non-numeric character is encountered in the string. The character that caused parsing to end is provide in *terminator*. You may set *terminator* to `NULL` if you do not care about the character that caused parsing to terminate.

Important:

This function will alter the contents of the string being parsed.

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>output</i> or <i>start</i> is <code>NULL</code> .
<code>AW_ERROR_FORMAT</code>	The first non-space character encountered in <i>string</i> was not a digit.

See also:

“awParseNumberFromString” on page 311

“awParseStringFromString” on page 312

awParseNumberFromString

```
BrokerError awParseNumberFromString(
    char *start,
```

```
char **next,
char *terminator,
int *output);
```

<i>start</i>	The string to be parsed.
<i>next</i>	The character immediately following the character that caused parsing to terminate. This parameter is used for output.
<i>terminator</i>	The character that caused parsing to terminate. This parameter is used for output.
<i>output</i>	The extracted value. This parameter is used for output.

Extracts a `int` type from the string specified by *start*. Parsing will proceed until a non-numeric character is encountered in the string. The character that caused parsing to end is provided in *terminator*. The character immediately following the character that caused parsing to terminate is provided in *next*.

You may set *terminator* to `NULL` if you do not need to know which character caused parsing to terminate. You may set *next* to `NULL` if you do not need to know the character that follows the terminating character.

Important:

This function will alter the contents of the string being parsed.

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAM	The parameter <i>output</i> or <i>start</i> is <code>NULL</code> .
AW_ERROR_FORMAT	The first non-space character encountered in <i>string</i> was not a digit.

See also:

“`awParseLongFromString`” on page 311

“`awParseStringFromString`” on page 312

awParseStringFromString

```
BrokerError awParseStringFromString(
char *start,
char **next,
int do_copy,
char **output);
```

<i>start</i>	The string to be parsed.
--------------	--------------------------

<i>next</i>	The character immediately following the character that caused parsing to terminate. This parameter is used for output.
<i>do_copy</i>	Indicates whether or not memory should be allocated for the extracted string. If set to 1 (<code>true</code>), memory will be allocated.
<i>output</i>	The extracted value. This parameter is used for output.

Extracts a string from the source string specified by *start*. Parsing will proceed until a null or newline character is encountered. The character immediately following the one that caused parsing to end is provided in *next*. You may set *next* to NULL if you do not need to know about this character.

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAM	The parameter <i>output</i> or <i>start</i> are NULL.

Important:

This function will alter the contents of the string being parsed.

See also:

“[awParseLongFromString](#)” on page 311

“[awParseNumberFromString](#)” on page 311

awPrime

awPrimeAllClients

```
BrokerError awPrimeAllClients();
```

Tells all Broker clients in your application to send a non-blocking request for events. You do not need to provide references to each of your clients. This function is intended to be used in conjunction with [awIsPending](#) and [awGetFds](#) to determine when events are ready to be received. Use this function when you are integrating webMethods Broker event handling with other event-based APIs.

Note:

Calling this function before invoking [awGetEvents](#) will cause only a single event to be returned, even if multiple events are available in the event queue.

See also:

“[awIsClientPending](#)” on page 280

“[awIsPending](#)” on page 283

“[awPrimeClient](#)” on page 314

awPrimeClient

```
BrokerError awPrimeClient(
    BrokerClient client);
```

client The client being primed.

Tells the specified *client* to send a non-blocking request for events. This function is intended to be used in conjunction with `awIsPending` and `awGetFds` to determine when events are ready to be received. Use this function when you are integrating webMethods Broker event handling with other event-based APIs.

Note:

Calling this function before invoking `awGetEvents` will cause only a single event to be returned, even if multiple events are available in the event queue.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.

See also:

“`awIsClientPending`” on page 280

“`awIsPending`” on page 283

“`awPrimeAllClients`” on page 313

awPublish

awPublishEvent

```
BrokerError awPublishEvent(
    BrokerClient client,
    BrokerEvent event)
```

client The Broker client that wishes to publish the event.

event The event to be published.

Sends the specified *event* to the Broker for publication.

Note:

This function can fail if the Broker client does not have permission to publish the event type, based on its client group, or if the event is not properly formed. See [awCanPublish](#) for more information.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_INVALID_EVENT	The <i>event</i> is not valid or the event does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish the event type.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awCanPublish” on page 153

“awDeliverErrorReplyEvent” on page 170

“awDeliverEvents” on page 173

“awPublishEvents” on page 315

“awPublishEventsWithAck” on page 316

awPublishEvents

```
BrokerError awPublishEvents (
    BrokerClient client,
    int n,
    BrokerEvent *events);
```

client The Broker client that wishes to publish the events.

n The number of events in the events array.

events The array of events to be published.

Sends the array of *events* to the Broker for publication. Either all events or none will be published.

Note:

This function can fail if the Broker client does not have permission to publish this event type, based on its client group, or if the event is not properly formed. See [awCanPublish](#) for more information.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_INVALID_EVENT	The <i>event</i> is not valid or the event does not match its type definition.

Possible BrokerError major codes	Meaning
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish all of the event type.
AW_ERROR_NULL_PARAM	The parameter <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awCanPublish” on page 153

“awDeliverErrorReplyEvent” on page 170

“awDeliverEvents” on page 173

“awPublishEvent” on page 314

“awPublishEventsWithAck” on page 316

awPublishEventsWithAck

```
BrokerError awPublishEventsWithAck(
    BrokerClient client,
    int n,
    BrokerEvent *events,
    int ack_type,
    int n_acks,
    BrokerLong *ack_seqn);
```

<i>client</i>	The Broker client that wishes to publish the events.
<i>n</i>	The number of events in the events array.
<i>events</i>	The array of events to be published.
<i>ack_type</i>	Determines how the events will be acknowledged and must be set to: AW_ACK_NONE, AW_ACK_AUTOMATIC, AW_ACK_THROUGH, or AW_ACK_SELECTIVE.
<i>n_acks</i>	The number of sequence numbers in <i>ack_seqn</i> .
<i>ack_seqn</i>	The array of sequence numbers to be acknowledged if <i>ack_type</i> is set to AW_ACK_THROUGH or AW_ACK_SELECTIVE.

Sends the array of *events* to the Broker for publication with one of several options for acknowledging events already received by this client. Either all events or none will be published.

Note:

This function can fail if the Broker client does not have permission to publish this event type, based on its client group, or if the event is not properly formed. See [awCanPublish](#) for more information.

The setting of the *ack_type* and *ack_seqn* parameters will determine which events received by this client are to be acknowledged.

ack_type	ack_seqn	Result
AW_ACK_NONE	Not applicable.	No events are acknowledged.
AW_ACK_AUTOMATIC	Not applicable.	All events received by the client are acknowledged.
AW_ACK_THROUGH	<i>ack_seqn[0]</i> contains the sequence number of the last event to be acknowledged. If set to 0, the behavior will be the same as AW_ACK_AUTOMATIC	Acknowledges all events up to and including the sequence number specified by <i>ack_seqn[0]</i> . If the <i>n_acks</i> argument is zero, no events will be acknowledged.
AW_ACK_SELECTIVE	<i>ack_seqn</i> contains the sequence numbers of the specific events to be acknowledged.	Acknowledges the specific events whose sequence numbers are contained in <i>ack_seqn</i> . The <i>n_acks</i> parameter must specify the number of sequence numbers contained in <i>ack_seqn</i> . All sequence numbers must be greater than zero.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_ACKNOWLEDGEMENT	The parameter <i>ack_seqn</i> contained an invalid sequence number. The events were not sent to the Broker.
AW_ERROR_INVALID_CLIENT	The <i>client</i> is not valid.
AW_ERROR_INVALID_EVENT	One of the events in <i>events</i> is not valid or the event does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish all of the event types.
AW_ERROR_NULL_PARAM	The parameter <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awCanPublish” on page 153

“awDeliverErrorReplyEvent” on page 170

“awDeliverEvents” on page 173

“awDeliverEventsWithAck” on page 174

“awPublishEvent” on page 314

“awPublishEvents” on page 315

awPublishRequestAndWait

```
BrokerError awPublishRequestAndWait(
    BrokerClient client,
    BrokerEvent event,
    int msecs,
    int *n,
    BrokerEvent **reply_events);
```

<i>client</i>	The Broker client that is sending the event.
<i>event</i>	The request event to be published.
<i>msecs</i>	The number of milliseconds to block awaiting a reply. If set to <code>AW_INFINITE</code> , this function will block indefinitely.
<i>n</i>	The number of reply events returned in the <i>events</i> array.
<i>reply_events</i>	The array of reply events that are returned.

Publishes the single *event* and then waits for all replies to be received. The last reply event is detected when an event is received with the envelope fields `appSeqn` and `appLastSeqn` being equal.

This function creates a value for the `tag` envelope field using [awMakeTag](#) function. This function blocks until the replies are received or until the requested time-out interval expires.

You are responsible for calling [awDeleteEvent](#) on each event that is returned. You are also responsible for calling `free` on the array itself.

See [“Using Request-Reply” on page 91](#) for more information on using the request-reply model.

Note:

You must register a general callback object, using the [awRegisterCallback](#) function, before calling this function.

Possible BrokerError major codes	Meaning
<code>AW_ERROR_BAD_STATE</code>	This function was called from a callback function.
<code>AW_ERROR_INVALID_CLIENT</code>	The <i>client</i> has been destroyed or disconnected.
<code>AW_ERROR_INVALID_EVENT</code>	The <i>event</i> is invalid.

Possible BrokerError major codes	Meaning
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish the event type.
AW_ERROR_NULL_PARAM	The parameter <i>n</i> or <i>reply_events</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awDeliverReplyEvents” on page 180

“awDeliverRequestAndWait” on page 181

“awPublishEvent” on page 314

awRealloc

awRealloc

```
oid *awRealloc(
    void *ptr,
    size_t size);
```

ptr Pointer to the memory block whose size is to be changed.

size The new size for the memory block.

Changes the size of the block of memory pointed to by *ptr*, which was previously allocated using the [awMalloc](#) function. A pointer to the new memory block is returned. This function returns a NULL pointer if additional memory could not be allocated.

If *size* is less than or equal to the previous size of the allocated memory block, any remaining memory contents will remain unchanged.

See also:

“awFree” on page 197

“awMalloc” on page 289

awReconnect

awReconnectBrokerClient

```
BrokerError awReconnectBrokerClient(
    char *broker_host,
```

```
char *broker_name,
char *client_id,
BrokerConnectionDescriptor desc,
BrokerClient *client);
```

<i>broker_host</i>	The name of the host running the Broker that the Broker client will use. Specified in the form <broker_host_name>:<port_number>. If the port number is omitted, the default port number will be used. For example: MyHost:12000
<i>broker_name</i>	The name of the Broker. If set to NULL, the default Broker name is used. If specified, the length of this parameter must be 1 to 255 ANSI characters or 1 to 42 Unicode characters.
<i>client_id</i>	The client identifier, identifying the Broker client as the one that was previously disconnected. See “ Client Identifiers ” on page 45 for more information.
<i>desc</i>	The connection descriptor to use for the Broker client. If set to NULL, a default connection will be established which may be shared with other clients.
<i>client</i>	A pointer to the Broker client. This parameter is used for output.

Reconnects a client. *broker_name* can be NULL to request the default Broker. *desc* can be NULL to request a default connection. The state sharing flag in the descriptor is ignored by this call. Whether or not the client shares state can only be set when making a new client. For normal clients, only one active connection can be made to the Broker for a given *client_id*, so an error can be returned on reconnect if the *client_id* is already in use. For clients with shared state, multiple reconnects to the *client_id* can be made, but an error can be returned if the share limit is exceeded.

The caller is responsible for calling `awDestroyClient` on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_BROKER_NOT_RUNNING	A Broker with <i>broker_name</i> could not be found on the specified <i>broker_host</i> .
AW_ERROR_CLIENT_CONTENTION	Another Broker client is using the specified <i>client_id</i> . For clients with shared state, this error is returned if the maximum number of reconnected clients has been exceeded.
AW_ERROR_COMM_FAILURE	A problem occurred during connection establishment.
AW_ERROR_HOST_NOT_FOUND	The <i>broker_host</i> could not be found.
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.
AW_ERROR_NO_PERMISSION	Permission to join the specified <i>client_group</i> was denied.

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAMETER	The <i>broker_host</i> or <i>client_id</i> is NULL.
AW_ERROR_SECURITY	A secure connection was attempted, but was rejected by the Broker.
AW_ERROR_UNKNOWN_BROKER_NAME	The specified <i>broker_name</i> does not exist. If <i>broker_name</i> is NULL, this indicates that there is no default Broker.
AW_ERROR_UNKNOWN_CLIENT_ID	The specified <i>client_id</i> does not exist on the Broker.
AW_ERROR_BASIC_AUTH_FAILURE	Basic authentication failed for the user.
AW_ERROR_INVALID_BASICAUTH_VALUE	You did not specify a password for the basic authentication user.

See also:

“awDestroyClient” on page 183

“awDisconnectClient” on page 184

“awNewBrokerClient” on page 291

“awNewOrReconnectBrokerClient” on page 298

awReconnectBrokerTxClient

```
BrokerError awReconnectBrokerTxClient(
    char *broker_host,
    char *broker_name,
    char *client_id,
    BrokerConnectionDescriptor desc,
    BrokerTxClient *txclient);
```

<i>broker_host</i>	The name of the host which is running the Broker that the Transactional Broker client will use. Specified in the form <broker_host_name>:<port_number>. If you omit the port number, the default port number will be used. For example: MyHost:12000
<i>broker_name</i>	The name of the Broker to which the Transactional Broker client will be connected. If NULL, the default Broker will be used. If specified, the length of this parameter must be 1 to 255 ANSI characters or 1 to 42 Unicode characters.
<i>client_id</i>	A string containing a unique identifier for creating or reconnecting the Transactional Broker client. See “ Client Identifiers ” on page 45 for more information. The client id can be NULL to request the Broker to create an identifier (usually used with destroy-on-disconnect clients).

<i>desc</i>	The connection descriptor to use for the Transactional Broker client. If NULL, a default connection will be established for the new client, which may be shared with other Broker clients.
<i>txclient</i>	A pointer to the reconnected or newly created Transactional Broker client. This parameter is used for output.

Reconnects a transactional client. *broker_name* can be NULL to request the default Broker. *desc* can be NULL to request a default connection. The state sharing flag in the descriptor is ignored by this call. Whether or not the client shares state can only be set when making a new transactional client. For normal clients, only one active connection can be made to the Broker for a given *client_id*, so an error can be returned on reconnect if the *client_id* is already in use. For clients with shared state, multiple reconnects to the *client_id* can be made, but an error can be returned if the share limit is exceeded.

The caller is responsible for calling `awDestroyClient` on the output value.

Possible BrokerError major codes	Meaning
AW_ERROR_BROKER_NOT_RUNNING	The host exists but no Broker is running on that host.
AW_ERROR_CLIENT_CONTENTION	Another Broker client is using the specified <i>client_id</i> . For clients with shared state, this error is returned if the maximum number of reconnected clients has been exceeded.
AW_ERROR_COMM_FAILURE	A problem occurred during connection establishment.
AW_ERROR_HOST_NOT_FOUND	The <i>broker_host</i> could not be found.
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.
AW_ERROR_NO_PERMISSION	Permission to join the specified <i>client_group</i> was denied.
AW_ERROR_NULL_PARAM	The <i>broker_host</i> , <i>client_group</i> , <i>app_name</i> , or <i>client</i> parameter is NULL.
AW_ERROR_SECURITY	A secure connection was attempted, but was rejected by the Broker.
AW_ERROR_UNKNOWN_BROKER_NAME	The specified <i>broker_name</i> does not exist. If <i>broker_name</i> is NULL, this indicates that there is no default Broker.
AW_ERROR_UNKNOWN_CLIENT_ID	The specified <i>client_id</i> does not exist on the Broker.

awRegister

awRegisterCallback

```
BrokerError awRegisterCallback(
```

```
BrokerClient client,
BrokerCallback func,
void *client_data);
```

<i>client</i>	The Broker client for which the callback is being registered.
<i>func</i>	The callback function to be called to handle the event. See “Defining a Callback Function” on page 77 .
<i>client_data</i>	A pointer to user-defined data that will be passed to the callback function when it is invoked.

Registers a non-specific callback *func* to process any events received by *client*. If this function has already been called, the previous callback registration will be replaced with the new registration.

When an event is received for *client*, the callback function *func* will be invoked to process the event and will be passed *client_data* as a parameter. The content and use of *client_data* is determined by you the caller; it is not examined or used in any way by the Broker system.

Callbacks registered with this function are called *non-specific callbacks* because they may be invoked to handle any event. You may also use the `awRegisterCallbackForSubId` function, described on [awRegisterCallbackForSubId](#), to register a *specific callback* function for a particular event subscription ID. *Specific callback functions* will take precedence over a *non-specific callback function*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>func</i> is NULL.

See also:

“awCancelCallbackForSubId” on page 154

“awCancelCallbackForTag” on page 155

“awCancelCallbacks” on page 155

“awRegisterCallback” on page 322

“awRegisterCallbackForSubId” on page 323

“awRegisterCallbackForTag” on page 324

awRegisterCallbackForSubId

```
BrokerError awRegisterCallbackForSubId(
    BrokerClient client,
    int sub_id,
    BrokerCallback func,
    void *client_data);
```

<i>client</i>	The Broker client for which the callback is being registered.
<i>sub_id</i>	The specific subscription ID of the events to be handled.
<i>func</i>	The callback function to be called to handle the event. See “Defining a Callback Function” on page 77 .
<i>client_data</i>	A pointer to user-defined data that will be passed to the callback function when it is invoked.

Registers a specific callback *func* to process events with the specified *sub_id* that are received by *client*. When an event with the specified subscription ID is received for *client*, the callback function *func* will be invoked to process that event and will be passed *client_data* as a parameter. The content and use of *client_data* is determined by you the caller; it is not examined or used in any way by the Broker system.

Note:

You must register a general callback object, using the `awRegisterCallback` function, before calling this function.

If a callback function has already been registered for *sub_id*, the previous callback registration will be replaced with the new registration.

Note:

When using this function, make sure that each of the Broker client's event subscriptions uses a unique subscription ID if you want the events to be processed by different callback functions. Use non unique subscription IDs if you want multiple event types to be processed by the same callback function.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	No general callback has been registered.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>func</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>sub_id</i> parameter is less zero.

See also:

“`awCancelCallbackForSubId`” on page 154

“`awCancelCallbacks`” on page 155

“`awRegisterCallback`” on page 322

awRegisterCallbackForTag

```
BrokerError awRegisterCallbackForTag(
```

```
BrokerClient client,
int tag,
BrokerBoolean cancel_when_done,
BrokerCallback func,
void *client_data);
```

<i>client</i>	The Broker client for which the callback is being registered.
<i>tag</i>	The event tag field associated with the callback.
<i>cancel_when_done</i>	If set to 1 (true), the callback will be automatically cancelled when <code>awlsLastReplyEvent</code> returns 1 (true).
<i>func</i>	The callback function to be called to handle the event. See “Defining a Callback Function” on page 77 .
<i>client_data</i>	A pointer to user-defined data that will be passed to the callback function when it is invoked.

Registers a callback function for a specified *client* and event *tag*. All events with the specified tag will be sent to the callback function *func*. If *cancel_when_done* is set to true (1), then the callback will be automatically cancelled when the `awlsLastReplyEvent` function returns true. Otherwise, the callback will stay in effect until explicitly cancelled.

Note:

You must register a general callback object, using the `awRegisterCallback` function, before calling this function.

The tag envelope field will only be set for reply events which are sent to your client in response to a request event bearing the same tag field. These reply events are *delivered* to your client.

If a callback has already been registered for the specified tag, it will be replaced with the new callback.

If you want to have different event types handled by their own unique callback, be sure that each event subscription uses a unique subscription identifier.

Possible BrokerError major codes	Meaning
AW_ERROR_BAD_STATE	No general callback has been registered.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>func</i> is NULL.

See also:

“`awCancelCallbackForTag`” on page 155

“`awCancelCallbacks`” on page 155

“`awRegisterCallback`” on page 322

awRegisterClientConnectionCallback

```
BrokerError awRegisterClientConnectionCallback(  
    BrokerClient client,  
    BrokerConnectionCallback func,  
    void *client_data);
```

<i>client</i>	The client for whom the callback function is being registered.
<i>func</i>	The callback function will be invoked when this client is disconnected or reconnected. This may be set to <code>NULL</code> to remove a callback function previously registered for the client.
<i>client_data</i>	Data that is to be passed to the callback method. This may be set to null.

Registers a connection callback function *func* for the *client* that will be invoked if this client is disconnected or reconnect. If a callback function was already registered for this client, it will be replaced by the new callback function.

You may un-register a previously registered callback function by invoking this function with a null *func* parameter.

awRegisterTxClientConnectionCallback

```
BrokerError awRegisterTxClientConnectionCallback(  
    BrokerTxClient txclient,  
    BrokerConnectionCallback func,  
    void *client_data);
```

<i>txclient</i>	The transactional client for whom the callback function is being registered.
<i>func</i>	The callback function will be invoked when this transactional client is disconnected or reconnected. This may be set to <code>NULL</code> to remove a callback function previously registered for the transactional client.
<i>client_data</i>	Data that is to be passed to the callback method. This may be set to null.

Registers a connection callback function *func* for the *txclient* that will be invoked if this client is disconnected or reconnect. If a callback function was already registered for this client, it will be replaced by the new callback function.

You may un-register a previously registered callback function by invoking this function with a null *func* parameter.

The callback function will be called with `connect_state` set to the following:

<code>AW_CONNECT_STATE_DISCONNECTED</code>	If the client is disconnected.
<code>AW_CONNECT_STATE_CONNECTED</code>	If the client connection is re-established (only happens if automatic reconnect is enabled).

AW_CONNECT_STATE_RECONNECTED	If the disconnect is discovered, but the connection is re-established immediately (only happens if automatic reconnect is enabled).
------------------------------	-------------------------------------------------------------------------------------------------------------------------------------

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.

awResend

awResendUnacknowledgedEvents

```
BrokerError awResendUnacknowledgedEvents(
    BrokerClient client,
    int n_seqn_num,
    BrokerLong *seqn,
    BrokerEvent **events);
```

Resend unacknowledged events with receipt sequence numbers as specified. A list of seqn numbers are passed in as argument.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_ACKNOWLEDGENT	The <i>seqn</i> is not a valid event to acknowledge.
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_OUT_OF_RANGE	The parameter <i>seqn</i> is less than zero.

In addition, any of the communications errors can occur.

awSet

awSet<type>Field

The `awSet<type>Field` functions have a general syntax:

```
BrokerError awSet<type>Field(
    BrokerEvent event,
    char *field_name,
    <Field_Value_Type> value);
```

<i>event</i>	The event whose field is to be set.
<i>field_name</i>	Name of the destination event field to set.

value The source value for the field. The field value is different depending on the function as follows:

- `awSetBooleanField` has a field value type of `BrokerBoolean`.
- `awSetByteField` has a field value type of `char`.
- `awSetCharField` has a field value type of `char`.
- `awSetDateField` has a field value type of `BrokerDate`.
- `awSetDoubleField` has a field value type of `double`.
- `awSetFloatField` has a field value type of `float`.
- `awSetIntegerField` has a field value type of `int`.
- `awSetLongField` has a field value type of `BrokerLong`.
- `awSetLongFieldNative` has a field value type of `NativeLong`. (Available only on platforms which support 64-bit integral representations.)
- `awSetShortField` has a field value type of `short`.
- `awSetStringField` has a field value type of `char *`.
- `awSetUCCharField` has a field value type of `charUC`.
- `awSetUCStringField` has a field value type of `charUC *`.

The `awSet<type>Field` function sets the destination event field *field_name* to the source *value*. Event field values may be set in any order. To set an entire sequence field in a single function call, use `awSet<type>SeqField`.

See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

The following sample code sets two event fields, one with a float value and the other with a string value:

```
err = awSetFloatField (event, "xfield", x)
err = awSetStringField (event, "ssfield", "The value")
```

Note:

Attempting to set an event field with a type that does not match the event's definition will result in an error being returned. No error will be returned if `awSetStringField` is used to set a field whose type is `FIELD_TYPE_UNICODE_STRING`. In these cases, the ANSI string that is supplied to `awSetStringField` will automatically be converted to a unicode string.

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_FIELD_NOT_FOUND</code>	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type, such as using a subscript on a non-sequence field.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> is NULL.

See also:

“awSet<type>SeqField” on page 329

“awSetField” on page 345

“awSetStringFieldToSubstring” on page 352

“awSetStructFieldFromEvent” on page 353

“awSetUCStringFieldAsA” on page 357

“awSetUCStringFieldAsUTF8” on page 358

awSet<type>SeqField

The awSet<type>SeqField functions have a general syntax:

```
BrokerError awSet<type>SeqField(
    BrokerEvent event,
    char *field_name,
    int src_offset, int dest_offset,
    int n,
    <Sequence_Value_Type> value_array);
```

<i>event</i>	The event whose sequence field is to be set.
<i>field_name</i>	Name of the destination event field.
<i>src_offset</i>	Elements to skip from the beginning of the source elements.
<i>dest_offset</i>	Elements to skip from the beginning of the sequence.
<i>n</i>	Number of source elements.
<i>value_array</i>	The source value array. The sequence value type is different depending on the function as follows <ul style="list-style-type: none"> ■ awSetBooleanSeqField has a sequence value type of BrokerBoolean *. ■ awSetByteSeqField has a sequence value type of char *.

- `awSetCharSeqField` has a sequence value type of `char *`.
- `awSetDateSeqField` has a sequence value type of `BrokerDate *`.
- `awSetDoubleSeqField` has a sequence value type of `double *`.
- `awSetFloatSeqField` has a sequence value type of `float *`.
- `awSetIntegerSeqField` has a sequence value type of `int *`.
- `awSetLongSeqField` has a sequence value type of `BrokerLong *`.
- `awSetShortSeqField` has a sequence value type of `short *`.
- `awSetStringSeqField` has a sequence value type of `char **`.
- `awSetUCCharSeqField` has a sequence value type of `charUC *`.
- `awSetUCStringSeqField` has a sequence value type of `charUC **`.

Each `awSet<type>SeqField` function sets the destination sequence field, *field_name*, contained in *event* to the sequence of values contained in *value_array*. To set a non-sequence field in an event, use one of the [awSet<type>Field](#) functions.

Each of the `awSet<type>SeqField` functions may overwrite all or part of the destination sequence field. These functions may also cause the destination sequence to grow, if a larger number of elements are stored into the sequence. These functions *never* reduce the number of elements in the destination sequence: You must use `awSetSequenceFieldSize` function to reduce the size of a sequence. See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

Note:

Attempting to set an event field with a type that does not match the event's definition will result in an error being returned. No error will be returned if `awSetStringSeqField` is used to set a field whose type is `FIELD_TYPE_UNICODE_STRING`. In these cases, the ANSI string that is supplied to `awSetStringSeqField` will automatically be converted to a unicode string.

Possible BrokerError major codes	Meaning
<code>AW_ERROR_FIELD_NOT_FOUND</code>	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
<code>AW_ERROR_FIELD_TYPE_MISMATCH</code>	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type.
<code>AW_ERROR_INVALID_EVENT</code>	The <i>event</i> is invalid.
<code>AW_ERROR_INVALID_FIELD_NAME</code>	The <i>field_name</i> is invalid.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>field_name</i> or <i>value</i> is NULL.
<code>AW_ERROR_OUT_OF_RANGE</code>	The <i>src_offset</i> is out of range, the <i>dest_offset</i> is less than zero, or <i>n</i> is less than zero.

See also:

“awSet<type>Field” on page 327

“awSetField” on page 345

“awSetSequenceField” on page 349

“awSetStructSeqFieldFromEvents” on page 353

“awSetUCStringSeqFieldAsUTF8” on page 360

awSetBooleanField

See [awSet<type>Field](#).

awSetBooleanSeqField

See [awSet<type>SeqField](#).

awSetBrokerLong

```
void awSetBrokerLong(  
    BrokerLong *bl,  
    int high,  
    int low);
```

<i>bl</i>	The BrokerLong to be set.
<i>high</i>	The value to set for the high-order word.
<i>low</i>	The value to set for the low-order word.

Sets the specified `BrokerLong` using the two, 32-bit representations provided.

The parameter *high* represents the most significant 32 bits. The parameter *low* represents the least significant 32 bits.

See also:

“awBrokerToNativeLong” on page 152

“awGetBrokerLong” on page 203

“awNativetoBrokerLong” on page 290

awSetByteField

See [awSet<type>Field](#).

awSetByteSeqField

See [awSet<type>SeqField](#).

awSetCharField

See [awSet<type>Field](#).

awSetCharSeqField

See [awSet<type>SeqField](#).

awSetClientAutomaticControlLabel

```
BrokerError awSetClientAutomaticControlLabel(  
    BrokerClient client,  
    BrokerBoolean enabled);
```

<i>client</i>	The Broker client whose automatic control label is to be set.
<i>enabled</i>	If set to 1 (true), the Broker client's access label will be automatically set in the <code>controlLabel</code> envelope field of any event the client publishes or delivers.

Enables or disables the automatic insertion of this Broker client's access label into the `controlLabel` envelope field of any event the client publishes or delivers.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.

See also:

“[awGetClientAccessLabel](#)” on page 207

awSetClientInfoSet

```
BrokerError awSetClientInfoSet(  
    BrokerClient client,  
    BrokerEvent infoSet);
```

<i>client</i>	The Broker client whose infoSet is to be set.
<i>infoSet</i>	The infoSet to set for this client.

Set the *infoset* for the specified *client*. The *infoset* is specified as a Broker event for convenience. Each client can store one *infoset* that can be used to contain information about its state or configuration.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>infoset</i> is NULL.

See also:

“awGetClientInfoset” on page 212

awSetClientStateShareLimit

```
BrokerError awSetClientStateShareLimit(
    BrokerClient client,
    int limit);
```

client The Broker client whose share limit is to be set.

limit The maximum number of clients that may share state with *client*. If set to -1, an unlimited number of clients may share state with *client*.

Sets the maximum number of Broker clients that can share state with *client*. A client's state consists of its event queue and its event subscriptions. If *limit* is less than the current number of clients sharing the state, all of the Broker clients are allowed to remain connected. However, no new clients will be allowed to share the client state until the number of clients drops below *limit*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>client</i> has been destroyed or disconnected.
AW_ERROR_NO_PERMISSION	The <i>client</i> has an unshared client state.
AW_ERROR_OUT_OF_RANGE	The parameter <i>limit</i> is zero or less than -1.

See also:

“awGetClientStateShareLimit” on page 215

awSetCurrentError

```
void awSetCurrentError(BrokerError err);
```

err The error to set.

Sets the current error to *err*. Broker has one global error per thread which the system considers to be the current error. The system manages the memory associated with the current error.

See also:

“awCopyError” on page 166

awSetDate

```
void awSetDate(  
    BrokerDate *d,  
    int yr,  
    int mo,  
    int dy);
```

<i>d</i>	The date to be set.
<i>yr</i>	The four-digit year.
<i>mo</i>	One- or two-digit month.
<i>dy</i>	One- or two-digit day.

Initializes specified *BrokerDate* *d* to the specified *yr*, *mo*, and *day*. The time fields of the date will remain unaffected.

See also:

“awSetDateCtime” on page 334

“awSetDateTime” on page 335

“awSetTime” on page 354

awSetDateCtime

```
void awSetDateCtime(  
    BrokerDate *d,  
    time_t t);
```

<i>d</i>	The <i>BrokerDate</i> whose time is to be set.
<i>t</i>	The time.

Initializes the *BrokerDate* *d* using the specified *time_t* parameter *t*. The *BrokerDate* field *is_date_and_time* is set to 1 (true). For further information on *time_t*, refer to the standard C date and time functions in <time.h>.

See also:

“awSetDate” on page 334

“awSetDateTime” on page 335

“awSetTime” on page 354

awSetDateField

See [awSet<type>Field](#).

awSetDateSeqField

See [awSet<type>SeqField](#).

awSetDateTime

```
void awSetDateTime(
    BrokerDate *d,
    int yr,
    int mo,
    int dy,
    int hr,
    int m,
    int s,
    int ms);
```

<i>d</i>	The BrokerDate whose date and time is to be set.
<i>yr</i>	The four-digit year, such as 1996.
<i>mo</i>	The one or two-digit month.
<i>dy</i>	The one or two-digit day.
<i>hr</i>	The hour in 24 hour format (0 through 23)
<i>m</i>	The one or two-digit minute.
<i>s</i>	The one or two-digit second.
<i>ms</i>	The millisecond (0 through 999).

Sets the specified BrokerDate *d* to the specified date and time parameters. The BrokerDate field `is_date_and_time` is set to 1 (true) .

See also:

“awSetDate” on page 334

“awSetDateCtime” on page 334

“awSetDateTime” on page 335

awSetDefaultClientTimeout

```
int awSetDefaultClientTimeout(  
    int msec);
```

msec The time-out interval, in milliseconds.

Sets the default time-out for non-event Broker requests to the number of milliseconds specified by *msec*. The previous time-out value, in milliseconds, is returned.

The default time-out is 30 seconds. Setting the time-out to a lower value in high-performance environments can provide your client applications with a more timely indication that a failure may have occurred.

awSetDescriptorAccessLabelHint

```
BrokerError awSetDescriptorAccessLabelHint(  
    BrokerConnectionDescriptor desc,  
    char *hint);
```

desc The Broker descriptor. This parameter is used for output.

hint The access label hint to be set for this descriptor. May be set to NULL.

Sets the access label hint for this descriptor.

When using the access label feature, a client may specify a hint string to be used to look up the client's access label. The access label look-up occurs only when creating a client. The hint string will be ignored when reconnecting a Broker client.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.

See also:

“awGetClientAccessLabel” on page 207

awSetDescriptorAuthInfo

```
BrokerError awSetDescriptorAuthInfo(  
    BrokerConnectionDescriptor desc,  
    const char *auth_username,  
    const char *auth_password);
```

desc The Broker connection descriptor on which you want to set the basic authentication user name and password.

auth_username The basic authentication user name to be set on the Broker connection descriptor.

auth_password The basic authentication password to be set on the Broker connection descriptor.

Sets the basic authentication information on the descriptor, that is user name and password.

Note:

If you set a NULL value for the *auth_username* and *auth_password* parameters, basic authentication information is reset for this connection descriptor.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.
AW_ERROR_NULL_PARAM	One of the parameters, <i>auth_username</i> or <i>auth_password</i> , is NULL.
AW_ERROR_NO_MEMORY	Cannot allocate memory.

See also:

“[awGetDescriptorAuthUserName](#)” on page 218

awSetDescriptorAutomaticReconnect

```
BrokerError awSetDescriptorAutomaticReconnect(
    BrokerConnectionDescriptor desc,
    BrokerBoolean reconnect);
```

desc The Broker descriptor. This parameter is used for output.

reconnect If set to 1 (`true`), the automatic reconnect behavior is enabled for this descriptor. Otherwise, it is disabled.

Enables or disables the automatic reconnection feature for this descriptor, based on the setting of the *reconnect* parameter.

Note:

You can disable the automatic reconnection feature by explicitly invoking `awDisconnectClient` or `awDestroyClient`.

If automatic reconnection is enabled, the Broker client associated with this descriptor will be automatically reconnected if the connection to the Broker is lost.

See “[Automatic Reconnection](#)” on page 54 for a complete discussion of the automatic reconnect feature.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.

See also:

“awGetDescriptorAutomaticReconnect” on page 219

awSetDescriptorAutomaticRedeliveryCount

```
BrokerError awSetDescriptorAutomaticRedeliveryCount(
    BrokerConnectionDescriptor desc,
    BrokerBoolean auto_on);
```

desc The Broker descriptor for which automatic redelivery count is to be set.

auto_on If set to 1 (*true*), the automatic redelivery count is enabled for this descriptor. Otherwise, it is disabled.

Enables or disables the automatic redelivery count for this descriptor, based on the setting of the *auto_on* parameter.

If automatic redelivery count is enabled, the Broker client associated with this descriptor will be automatically maintain a redelivery count for each document in a queue.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> parameter is invalid.

See also:

“awGetDescriptorAutomaticRedeliveryCount” on page 219

awSetDescriptorConnectionShare

```
BrokerError awSetDescriptorConnectionShare(
    BrokerConnectionDescriptor desc,
    BrokerBoolean shared);
```

desc The connection descriptor to be set.

shared Determines whether or not this connection to the Broker can be shared by more than client. If set to 0 (*false*), the connection may not be shared. Any other non-zero value means the connection may be shared.

Sets connection share attribute for the descriptor *desc* to the value of *shared*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The descriptor <i>desc</i> is not valid.

See also:

“awGetDescriptorConnectionShare” on page 220

“awNewBrokerClient” on page 291

“awReconnectBrokerClient” on page 319

awSetDescriptorForcedReconnect

```
BrokerError awSetDescriptorForcedReconnect(
    BrokerConnectionDescriptor desc,
    BrokerBoolean forced_reconnect);
```

desc The Broker descriptor for which forced reconnect is to be set.

forced_reconnect Specifies whether a client program can forcibly reconnect to a Broker even if it appears that the client program is already connected to the Broker.

If set to 1 (`true`), forced reconnect is enabled for this descriptor. Otherwise, it is disabled.

Sets forced reconnect status for the specified descriptor.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.

See also:

“awGetDescriptorForcedReconnect” on page 220

awSetDescriptorKeepAlive

```
BrokerError awSetDescriptorKeepAlive(
    BrokerConnectionDescriptor desc,
    int KeepAliveSendPeriod,
    int MaxKeepAliveResponseTime,
    int RetryCount);
```

desc The Broker descriptor for which keep-alive is to be set.

<i>KeepAliveSendPeriod</i>	Specifies the length of time (in seconds) the Broker should wait before issuing the keep-alive messages on an idle connection.
<i>MaxKeepAliveResponseTime</i>	Specifies the length of time (in seconds) the Broker should wait for a reply from an idle connection.
<i>RetryCount</i>	Specifies how many times to send the keep-alive messages before disconnecting the unresponsive client.

Sets the keep-alive settings for the specified descriptor.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.

See also:

“awGetDescriptorKeepAlive” on page 221

awSetDescriptorRedeliveryCountEnabled

```
BrokerError awSetDescriptorRedeliveryCountEnabled(  
    BrokerConnectionDescriptor desc,  
    BrokerBoolean redelivery_count);
```

<i>desc</i>	The Broker descriptor for which the redelivery count is to be set.
<i>redelivery_count</i>	If set to <code>true</code> , enables redelivery counting. If set to <code>false</code> , disables redelivery counting.

Sets the redelivery counting status for the specified descriptor.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.

See also:

“awGetDescriptorRedeliveryCountEnabled” on page 222

awSetDescriptorSharedEventOrdering

```
BrokerError awSetDescriptorSharedEventOrdering(  
    BrokerConnectionDescriptor desc,  
    Char *ordering);
```

<i>desc</i>	The Broker descriptor whose shared event ordering is to be set.
<i>ordering</i>	The desired shared event ordering state, which is specified as either <code>AW_SHARED_ORDER_NONE</code> or <code>AW_SHARED_ORDER_BY_PUBLISHER</code> .

Sets the shared event ordering status for the specified descriptor. See [By-Publisher Event Ordering](#) for more information.

Possible BrokerError major codes	Meaning
<code>AW_ERROR_INVALID_DESCRIPTOR</code>	The <i>desc</i> is invalid.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>ordering</i> is NULL.
<code>AW_ERROR_OUT_OF_RANGE</code>	The parameter <i>ordering</i> contains spaces or non-printable characters.

See also:

“`awGetDescriptorSharedEventOrdering`” on page 222

awSetDescriptorSSLCertificate

```
BrokerError awSetDescriptorSSLCertificate(
    BrokerConnectionDescriptor desc,
    char *certificate_file,
    char *password,
    char *trust_file);
```

<i>desc</i>	The connection descriptor whose SSL information is to be provided.
<i>certificate_file</i>	The SSL certificate file to be associated with <i>desc</i> .
<i>password</i>	The SSL password for the certificate file.
<i>trust_file</i>	The SSL truststore file to be associated with <i>desc</i> .

Sets the secure socket layer (SSL) *certificate_file* and *trust_file* for *desc*.

For any Broker client created or reconnected with the descriptor *desc*:

- If the *certificate_file* is NULL, then SSL will be disabled.
- If *certificate_file* is not NULL and *trust_file* is NULL, then server-only authentication will be enabled.
- If both *certificate_file* and *trust_file* are not NULL, then both server and client authentication will be used.

The server must be properly configured to use SSL. See *Administering webMethods Broker* for complete information.

Possible BrokerError major codes	Meaning
AW_ERROR_FILE_NOT_FOUND	The <i>certificate_file</i> or <i>trust_file</i> could not be found or could not be read.
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.
AW_ERROR_NO_PERMISSION	The <i>password</i> is invalid or the <i>certificate_file</i> has an unrecognized format.
AW_ERROR_NULL_PARAM	The <i>password</i> is NULL when the <i>certificate_file</i> is not NULL.
AW_ERROR_SECURITY	SSL support is not available.

See also:

“awGetDescriptorSSLCertificate” on page 223

awSetDescriptorSSLEncrypted

```
BrokerError awSetDescriptorSSLEncrypted(
    BrokerConnectionDescriptor desc,
    BrokerBoolean encrypted);
```

desc The Broker descriptor whose SSL encryption state is to be returned.

encrypted If 0 (*false*), only SSL handshaking will be used. If 1 (*true*), all traffic on this descriptor will be encrypted.

Sets the value of the SSL encrypt flag for the descriptor *desc*. If *encrypted* is 0 (*false*), then data traffic for Broker clients created with the descriptor will not be encrypted once the SSL handshaking between server and client has been completed. If set to 1 (*true*), the default value, all data traffic that occurs after SSL handshaking will be encrypted.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The <i>desc</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>encrypted</i> is NULL.

See also:

“awGetDescriptorSSLEncrypted” on page 223

awSetDescriptorStateShare

```
BrokerError awSetDescriptorStateShare(
    BrokerConnectionDescriptor desc,
    BrokerBoolean shared);
```

<i>desc</i>	The connection descriptor to be set.
<i>shared</i>	Determines whether or not the Broker client associated with this descriptor can share its client state. If set to 0 (<i>false</i>), the client state may not be shared. Any other non-zero value means the client state may be shared.

Sets the client state sharing attribute for the descriptor *desc* to the value of *shared*. A Broker client's state consists of its event queue and its set of event subscriptions. When *shared* is set to true more than one Broker client can share this state. All Broker clients that share state have the same client ID.

Though events are placed on the event queue in the order that they are published, it is not predictable which of the Broker clients sharing that queue will receive a particular event.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_DESCRIPTOR	The descriptor <i>desc</i> is not valid.

See also:

“awGetClientStateShareLimit” on page 215

“awGetDescriptorSSLCertificate” on page 223

“awNewBrokerClient” on page 291

“awReconnectBrokerClient” on page 319

“awSetClientStateShareLimit” on page 333

awSetDiagnostics

```
void awSetDiagnostics(
    int diag);
```

diag The diagnostic level to set.

Sets the system diagnostic level to the value specified by *diag*. The diagnostic level determines the type of Broker error reporting that will occur for your application. The possible values and their meanings are:

Diagnostic Level	Description
0	No error output will be produced.
1	Produces error output for major errors only. This is the default setting
2	Produces all error output for all methods.

See also:

“awGetDiagnostics” on page 224

awSetDoubleField

See [awSet<type>Field](#).

awSetDoubleSeqField

See [awSet<type>SeqField](#).

awSetEventPublishSequenceNumber

```
BrokerError awSetEventPublishSequenceNumber(  
    BrokerEvent event,  
    BrokerLong seqn);
```

event The event whose sequence number is to be set.

seqn The sequence number to set.

Sets the publishing sequence number of the specified *event* to the value *seqn*. If your client wants to ensure the delivery of an event to the Broker, this function should be called to set the sequence number prior to publishing the event.

The Broker will discard any event with a non-zero sequence number if it has already processed an event with an equal or larger sequence number. Events with an event sequence number of zero are not subject to these rules. Sequence numbers have a 64-bit representation and, therefore, are never expected to wrap back to 1 or to repeat.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_EVENT	The <i>event</i> is not valid.

Note:

The `awSetEventPublishSequenceNumber` function does not actually set the `pubSeqn` envelope field. Instead, it specifies the sequence number that the Broker is to set on the next event published by your Broker client.

See also:

“awGetEventPublishSequenceNumber” on page 229

“awPublishEvent” on page 314

“awPublishEvents” on page 315

awSetEventTag

```
BrokerError awSetEventTag(
    BrokerEvent event,
    int tag);
```

Set the tag envelope field of *event* to the value represented by *tag*. This is equivalent to, but more convenient than, calling:

```
awSetIntegerField(event, "_env.tag", tag);
```

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.

See also:

“awGetEvents” on page 230

“awMakeTag” on page 285

awSetField

```
BrokerError awSetField(
    BrokerEvent event,
    char *field_name,
    short field_type,
    void *value);
```

<i>event</i>	The event whose field is to be set.
<i>field_name</i>	The name of the event field to set.
<i>field_type</i>	The field's type, such as FIELD_TYPE_BOOLEAN (see the table below).
<i>value</i>	The field's new value.

Sets the value of the destination event field *field_name* to the source *value*. To set sequence fields, use the `awSetSequenceField` function described on [awSetSequenceField](#). The use of the source *value* depends on the type of the destination field, indicated by *field_type*.

- For basic types, pass the address of a variable of that type.
- For strings, pass a `char*` pointer.
- For structures, pass a `BrokerEvent`.
- For `BrokerBoolean`, pass one of these two values: `True (1)` or `False (0)`.

Note:

Attempting to set a an event field with a type that does not match the event's definition will result in an error being returned. No error will be returned if a `FIELD_TYPE_STRING` value is used to set a field whose type is `FIELD_TYPE_UNICODE_STRING`. In these cases, the ANSI string that is supplied will automatically be converted to a unicode string.

See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

Field Type Values for `awSetField`

<code>FIELD_TYPE_BYTE</code>	<code>char *</code>
<code>FIELD_TYPE_SHORT</code>	<code>short*</code>
<code>FIELD_TYPE_INT</code>	<code>int *</code>
<code>FIELD_TYPE_LONG</code>	<code>BrokerLong *</code>
<code>FIELD_TYPE_FLOAT</code>	<code>float *</code>
<code>FIELD_TYPE_DOUBLE</code>	<code>double *</code>
<code>FIELD_TYPE_BOOLEAN</code>	<code>BrokerBoolean *</code>
<code>FIELD_TYPE_DATE</code>	<code>BrokerDate *</code>
<code>FIELD_TYPE_CHAR</code>	<code>char *</code>
<code>FIELD_TYPE_STRING</code>	<code>char *</code>
<code>FIELD_TYPE_STRUCT</code>	<code>BrokerEvent</code>
<code>FIELD_TYPE_UNICODE_CHAR</code>	<code>charUC *</code>
<code>FIELD_TYPE_UNICODE_STRING</code>	<code>charUC *</code>

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_FIELD_NOT_FOUND</code>	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
<code>AW_ERROR_FIELD_TYPE_MISMATCH</code>	The field's type does not match the type of <i>field_type</i> or the <i>field_name</i> incorrectly accesses a type.
<code>AW_ERROR_INVALID_EVENT</code>	The <i>event</i> is invalid.
<code>AW_ERROR_INVALID_FIELD_NAME</code>	The <i>field_name</i> is invalid.
<code>AW_ERROR_INVALID_TYPE</code>	The <i>field_type</i> is <code>FIELD_TYPE_SEQUENCE</code> or is not a supported field type.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>field_name</i> or <i>value</i> is <code>NULL</code> .

See also:

“[awSet<type>Field](#)” on page 327

“[awSet<type>SeqField](#)” on page 329

“[awSetSequenceField](#)” on page 349

“[awSetStringFieldToSubstring](#)” on page 352

awSetFloatField

See [awSet<type>Field](#).

awSetFloatSeqField

See [awSet<type>SeqField](#)

awSetFormatMode

```
BrokerBoolean awSetFormatMode(
    char *format_option,
    char *mode);
```

format_option The format option to be set.

mode The format mode to be set.

Sets the *mode* for the *format_option*. Returns 1 (true) if the mode was altered, otherwise 0 (false) is returned.

Consider the following format string:

```
column1 = ${field1:v}, column2 = ${field2:v}
```

If you can call this function as:

```
awSetFormatMode("v", "?")
```

Invoking `BrokerFormat.awEventFormatAssemble` will produce this output:

```
column1 = ?, column2 = ?
```

If you call this function as:

```
awSetFormatMode("v", ":")
```

Invoking `awEventFormatAssemble` will produce this output:

```
column1 = :v0, column2 = :v1
```

See also:

“[awEventFormatAssemble](#)” on page 190

“awEventFormatBindVariable” on page 191

“awEventFormatFree” on page 192

“awEventFormatPreparse” on page 192

“awEventToBinData” on page 194

“awEventFormatTokens” on page 193

awSetIntegerField

See [awSet<type>Field](#).

awSetIntegerSeqField

See [awSet<type>SeqField](#).

awSetlocale

```
char * awSetlocale(int category, char *locale);
```

This call is not necessary on most platforms or under most conditions. It is necessary on MS Windows environments when the calling application is using a different C runtime library than the API.

awSetLongField

See [awSet<type>Field](#).

awSetLongFieldNative

See [awSet<type>Field](#).

awSetLongSeqField

See [awSet<type>SeqField](#).

awSetPlatformInfo

```
BrokerError awSetPlatformInfo(  
    char *key,  
    char *value);
```

key The key to set.

value The value to associate with *key*.

Sets the value of a single platform information key. If the key-value pair doesn't exist, it is added to the platform information.

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAM	The <i>key</i> or <i>value</i> parameter is NULL.
AW_ERROR_NO_PERMISSION	The specified <i>key</i> is read-only.

See also:

“awGetPlatformInfo” on page 245

“awGetPlatformInfoKeys” on page 246

awSetSequenceField

```
BrokerError awSetSequenceField(
    BrokerEvent event,
    char *field_name,
    short field_type,
    int src_offset, int dest_offset,
    int n,
    void *source_values);
```

<i>event</i>	The event whose sequence field is to be set.
<i>field_name</i>	Name of the event sequence field.
<i>field_type</i>	A field type, such as FIELD_TYPE_BOOLEAN, defined in <awetdef.h>.
<i>src_offset</i>	The number of elements to skip from the beginning of the source sequence.
<i>dest_offset</i>	The number of elements to skip from the beginning of the destination sequence.
<i>n</i>	Number of elements in the source sequence.
<i>source_values</i>	An array of source values to use. See awSet<type>SeqField for the appropriate types.

Sets the values of the destination sequence field *field_name* to the values contained in the source sequence *source_values*. The elements in the source and destination sequence are of the type represented by *field_type*.

Note:

A sequence of sequences cannot be set with this function.

This function may overwrite all or part of the destination sequence field or cause the destination sequence to grow, if a larger number of elements are stored into the sequence. This function *never*

reduces the number of elements in the destination sequence: You must use `awSetSequenceFieldSize` function to reduce the size of a sequence.

If you set five elements (a, b, c, d, e) into a sequence at location 0, the sequence would appear as: [a b c d e]. If you then set three elements (1, 2, 3) into this same sequence at location 0, the sequence would then appear as follows:

```
[1 2 3 d e]
```

Note:

Attempting to set an event field with a type that does not match the event's definition will result in an error being returned. No error will be returned if `FIELD_TYPE_STRING` values are used to set a sequence field whose type is `FIELD_TYPE_UNICODE_STRING`. In these cases, the ANSI strings that are supplied will automatically be converted to unicode strings.

See [“Specifying Field Names” on page 37](#) for complete information on specifying `field_name`.

Possible BrokerError major codes	Meaning
<code>AW_ERROR_FIELD_NOT_FOUND</code>	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
<code>AW_ERROR_FIELD_TYPE_MISMATCH</code>	The field's type does not match the type of <i>field_type</i> or the <i>field_name</i> incorrectly accesses a type.
<code>AW_ERROR_INVALID_EVENT</code>	The <i>event</i> is invalid.
<code>AW_ERROR_INVALID_FIELD_NAME</code>	The <i>field_name</i> is invalid.
<code>AW_ERROR_INVALID_TYPE</code>	The <i>field_type</i> is <code>FIELD_TYPE_SEQUENCE</code> or is not a supported field type.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>field_name</i> or <i>source_value</i> is NULL.
<code>AW_ERROR_OUT_OF_RANGE</code>	The <i>src_offset</i> is out of range, the <i>dest_offset</i> is less than zero, or <i>n</i> is less than -1.

See also:

“`awGetSequenceField`” on page 248

“`awSet<type>SeqField`” on page 329

“`awSetStructSeqFieldFromEvents`” on page 353

awSetSequenceFieldSize

```
BrokerError awSetSequenceFieldSize(
    BrokerEvent event,
    char *field_name,
    int size);
```

<i>event</i>	The event whose sequence field size is to be set.
<i>field_name</i>	Name of the event sequence field.
<i>size</i>	The number of elements in the sequence.

Sets the *size* of the sequence field *field_name*. See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>size</i> is less than zero.

See also:

“[awGetSequenceField](#)” on page 248

“[awSet<type>SeqField](#)” on page 329

“[awSetSequenceField](#)” on page 349

“[awSetStructSeqFieldFromEvents](#)” on page 353

awSetShortField

See [awSet<type>Field](#).

awSetShortSeqField

See [awSet<type>SeqField](#).

awSetStringField

See [awSet<type>Field](#).

awSetStringFieldToSubstring

```
BrokerError awSetStringFieldToSubstring(
    BrokerEvent event,
    char *field_name,
    int char_offset,
    int nc,
    char *value);
```

<i>event</i>	The event whose string field is to be set.
<i>field_name</i>	Name of the string field.
<i>char_offset</i>	Character offset from beginning of field.
<i>nc</i>	Number of characters to use. If set to -1, the rest of the string will be used.
<i>value</i>	The new value for the string field.

Sets the value of the string field *field_name* to the specified sub-string *value*. The number of characters to be set is specified by *nc*. The offset into the field is specified by *char_offset*.

See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type is not a string or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> or <i>value</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>char_offset</i> is out of range, the <i>dest_offset</i> is less than zero, or <i>nc</i> is less than -1.

See also:

“awSetStringField” on page 351

“awSet<type>Field” on page 327

awSetStringSeqField

See [awSet<type>SeqField](#).

awSetStructFieldFromEvent

```
BrokerError awSetStructFieldFromEvent(
    BrokerEvent event,
    char *field_name,
    BrokerEvent value);
```

event The event whose structure field is to be set.

field_name Name of the event field structure whose values are being set. May be set to NULL if the top level of the event is to be set.

value The new value for *field_name*.

Sets the field specified by *field_name*, of type struct, from the specified *value*. The parameter *value* is an event that contains the structure-field values. Any envelope fields contained in *value* will be ignored.

See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.

See also:

“awSet<type>Field” on page 327

“awSetField” on page 345

“awSetStructSeqFieldFromEvents” on page 353

awSetStructSeqFieldFromEvents

```
BrokerError awSetStructSeqFieldFromEvents(
    BrokerEvent event,
    char *field_name,
    int src_offset,
    int dest_offset,
    int n, BrokerEvent *value);
```

event The event whose field is to be set.

<i>field_name</i>	The name of the event field to be set.
<i>src_offset</i>	The number of elements to skip from the beginning of the source elements.
<i>dest_offset</i>	The number of elements to skip from the beginning of the destination sequence.
<i>n</i>	Number of source elements (structures) contained in <i>value</i> .
<i>value</i>	An array of events that represent structures.

Sets the destination structure sequence field *field_name* from the specified source *value*, which is expressed as an array of events. Each of the events in *value* should contain fields that correspond to the fields of a structure. Envelope fields on the value events are ignored.

Note:

This function may overwrite all or part of the destination structure sequence field.

See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> or any event in the array <i>value</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> or <i>value</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>src_offset</i> is not within the range of the array <i>value</i> , the <i>dest_offset</i> is less than zero, or <i>n</i> is less than -1.

See also:

“awSet<type>SeqField” on page 329

“awSetSequenceField” on page 349

“awSetStructFieldFromEvent” on page 353

awSetTime

```
void awSetTime(
    BrokerDate *d,
    int hr,
    int m,
    int s,
```

```
int ms);
```

<i>d</i>	The BrokerDate whose date and time is to be set.
<i>hr</i>	The hour in 24 hour format (0 through 23)
<i>m</i>	The one or two-digit minute.
<i>s</i>	The one or two-digit second.
<i>ms</i>	The millisecond (0 through 999).

Sets the time portion of the BrokerDate *d* with the specified time parameters. It also sets `is_date_and_time` to 1 (true). The year, month, and day fields are not affected.

See also:

“awSetDate” on page 334

“awSetDateCtime” on page 334

“awSetDateTime” on page 335

awSetTxClientAutomaticControlLabel

```
BrokerError awSetTxClientAutomaticControlLabel(
    BrokerTxClient txclient,
    BrokerBoolean enabled);
```

<i>txclient</i>	The Broker client whose automatic control label is to be set.
<i>enabled</i>	If set to 1 (true), the Broker client's access label will be automatically set in the <code>controlLabel</code> envelope field of any event the client publishes or delivers.

Enables or disables the automatic insertion of this Broker client's access label into the `controlLabel` envelope field of any event the transactional client publishes or delivers.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.

See also:

“awGetTxClientAccessLabel” on page 256

awSetTxClientInfoSet

```
BrokerError awSetTxClientInfoSet(
```

```
BrokerTxClient txclient,
BrokerEvent infoSet);
```

txclient The Broker client whose infoSet is to be set.

infoSet The infoSet to set for this client.

Set the *infoSet* for the specified *txclient*. The infoSet is specified as a Broker event for convenience. Each client can store one infoSet that can be used to contain information about its state or configuration.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>infoSet</i> is NULL.

See also:

“awGetTxClientInfoSet” on page 261

awSetTxClientStateShareLimit

```
BrokerError awSetTxClientStateShareLimit(
BrokerTxClient txclient,
int limit);
```

txclient The Broker client whose share limit is to be set.

limit The maximum number of clients that may share state with *txclient*. If set to -1, an unlimited number of clients may share state with *txclient*.

Sets the maximum number of Broker clients that can share state with *txclient*. A client's state consists of its event queue and its event subscriptions. If *limit* is less than the current number of clients sharing the state, all of the Broker clients are allowed to remain connected. However, no new clients will be allowed to share the client state until the number of clients drops below *limit*.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> has an unshared client state.
AW_ERROR_OUT_OF_RANGE	The parameter <i>limit</i> is zero or less than -1.

See also:

“awGetTxClientStateShareLimit” on page 264

awSetUCCharField

See [awSet<type>Field](#).

awSetUCCharSeqField

See [awSet<type>SeqField](#).

awSetUCStringField

See [awSet<type>Field](#).

awSetUCStringFieldAsA

```
BrokerError awSetUCStringFieldAsA(
    BrokerEvent event,
    char *field_name,
    char *value);
```

<i>event</i>	The event whose field is to be set.
<i>field_name</i>	Name of the destination event field to set.
<i>value</i>	The source value for the field. The value's type varies from function to function.

Sets the specified event field using the *value*, which contains a string in ANSI format. ANSI format allows 16-bit Unicode characters to be represented using an escape notation, such as `\u1234`.

See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

Note:

Attempting to set an event field with a type that does not match the event's definition will result in an error being returned.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>value</i> or the <i>field_name</i> incorrectly accesses a type, such as using a subscript on a non-sequence field.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.

Possible BrokerError major codes	Meaning
AW_ERROR_NULL_PARAM	The <i>field_name</i> or <i>value</i> is NULL.

See also:

“awGetUCStringFieldAsA” on page 276

“awGetUCStringFieldAsUTF8” on page 277

“awSet<type>Field” on page 327

“awSet<type>SeqField” on page 329

“awSetField” on page 345

“awSetStringFieldToSubstring” on page 352

“awSetStructFieldFromEvent” on page 353

“awSetUCStringSeqFieldAsUTF8” on page 360

awSetUCStringFieldAsUTF8

```
BrokerError awSetUCStringFieldAsUTF8(
    BrokerEvent event,
    char *field_name,
    char *utf8_value);
```

<i>event</i>	The event whose field is to be set.
<i>field_name</i>	Name of the destination event field to set.
<i>value</i>	The source value for the field. The value's type varies from function to function.

Sets the specified event field using the *utf8_value*, which contains a string in Unicode Transform Function 8 (UTF-8) format. UTF-8 format allows 16-bit Unicode characters to be represented as a sequence of up to four 8-bit encoded characters.

See “[Specifying Field Names](#)” on page 37 for complete information on specifying *field_name*.

Note:

Attempting to set an event field with a type that does not match the event's definition will result in an error being returned.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>utf8_value</i> or the <i>field_name</i> incorrectly accesses a type, such as using a subscript on a non-sequence field.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The <i>field_name</i> or <i>utf8_value</i> is NULL.

See also:

- “awGetUCStringFieldAsA” on page 276
- “awGetUCStringFieldAsUTF8” on page 277
- “awSet<type>Field” on page 327
- “awSet<type>SeqField” on page 329
- “awSetField” on page 345
- “awSetStringFieldToSubstring” on page 352
- “awSetStructFieldFromEvent” on page 353

awSetUCStringFieldToSubstring

```
BrokerError awSetUCStringFieldToSubstring(
    BrokerEvent event,
    char *field_name,
    int char_offset,
    int nc,
    char *value);
```

<i>event</i>	The event whose string field is to be set.
<i>field_name</i>	Name of the string field.
<i>char_offset</i>	Character offset from beginning of field.
<i>nc</i>	Number of characters to use. If set to -1, the rest of the string will be used.
<i>value</i>	The new value for the string field.

Sets the value of the Unicode string field *field_name* to the specified sub-string *value*. The number of characters to be set is specified by *nc*. The offset into the field is specified by *char_offset*.

See [“Specifying Field Names” on page 37](#) for complete information on specifying *field_name*.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type is not a string or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> or <i>value</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>char_offset</i> is out of range, the <i>dest_offset</i> is less than zero, or <i>nc</i> is less than -1.

See also:

“awSetStringField” on page 351

“awSet<type>Field” on page 327

awSetUCStringSeqField

See [awSet<type>SeqField](#).

awSetUCStringSeqFieldAsUTF8

```
BrokerError awSetUCStringSeqFieldAsUTF8(
    BrokerEvent event,
    char *field_name,
    int src_offset, int dest_offset,
    int n,
    char **utf8_value);
```

<i>event</i>	The event whose sequence field is to be set.
<i>field_name</i>	Name of the destination event field.
<i>src_offset</i>	Elements to skip from the beginning of the source elements.
<i>dest_offset</i>	Elements to skip from the beginning of the sequence.
<i>n</i>	Number of source elements.
<i>utf8_value</i>	The source value array containing a sequence of UTF-8 encoded strings.

Sets the destination sequence field, *field_name*, contained in *event* to the sequence of values contained in *utf8_value*. Each string pointed to by the *utf8_value* array is assumed to be in Unicode Transform

Function 8 (UTF-8) format. UTF-8 format allows 16-bit Unicode characters to be represented as a sequence of up to four 8-bit encoded characters.

This function may overwrite all or part of the destination sequence field. This function may also cause the destination sequence to grow, if a larger number of elements are stored into the sequence. This function *never* reduces the number of elements in the destination sequence: You must use `awSetSequenceFieldSize` function to reduce the size of a sequence.

See [“Specifying Field Names” on page 37](#) for complete information on specifying `field_name`.

Note:

Attempting to set an event field with a type that does not match the event's definition will result in an error being returned.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_NOT_FOUND	The <i>event</i> is associated with a Broker client and <i>field_name</i> does not exist in the event.
AW_ERROR_FIELD_TYPE_MISMATCH	The field's type does not match the type of <i>utf8_value</i> or the <i>field_name</i> incorrectly accesses a type.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_INVALID_FIELD_NAME	The <i>field_name</i> is invalid.
AW_ERROR_NULL_PARAM	The parameter <i>field_name</i> or <i>utf8_value</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>src_offset</i> is out of range, the <i>dest_offset</i> is less than zero, or <i>n</i> is less than -1.

See also:

“`awGetUCStringSeqFieldAsUTF8`” on page 278

“`awSet<type>Field`” on page 327

“`awSet<type>SeqField`” on page 329

“`awSetField`” on page 345

“`awSetSequenceField`” on page 349

“`awSetStructSeqFieldFromEvents`” on page 353

“`awSetUCStringFieldAsUTF8`” on page 358

awSSL

awSSLCertificateToIndentedString

```
char *awSSLCertificateToIndentedString(  
    BrokerSSLCertificate *cert,  
    int indent);
```

cert The certificate to be converted to a string.

indent The number of 4-space indentations to be generated in the output.

Returns a pointer to a string containing a human-readable version of the specified SSL certificate

Returns NULL if *cert* is invalid or if memory could not be allocated.

The caller is responsible for freeing the memory associated with the returned value.

See also:

“awSSLCertificateToString” on page 362

awSSLCertificateToString

```
char *awSSLCertificateToString(  
    BrokerSSLCertificate *cert);
```

cert The certificate to be converted to a string.

Returns a pointer to a string containing a human-readable version of the specified SSL certificate

Returns NULL if *cert* is invalid or if memory could not be allocated.

The caller is responsible for freeing the memory associated with the returned value.

See also:

“awSSLCertificateToIndentedString” on page 362

awStop

awStopMainLoop

```
BrokerError awStopMainLoop();
```

Interrupts the current [awMainLoop](#). If [awMainLoop](#) has not yet been called, the next call to [awMainLoop](#) will be interrupted. You may call this function from within callbacks or from multi-threaded clients. This function is also safe for use in a signal handler.

See also:

“[awMainLoop](#)” on page 284

awString

awStringAppend

```
void awStringAppend(  
    BrokerString st,  
    char *append);
```

st The destination string.

append The source string.

Appends the source string *append* to the end of the destination string *st*.

See also:

“[awStringAppendByte](#)” on page 363

“[awStringAppendChar](#)” on page 364

“[awStringAppendFloat](#)” on page 365

“[awStringAppendInteger](#)” on page 366

“[awStringAppendDouble](#)” on page 364

“[awStringAppendLong](#)” on page 366

“[awStringAppendNString](#)” on page 366

“[awStringAppendShort](#)” on page 367

“[awStringAppendNUCString](#)” on page 367

“[awStringAppendUCChar](#)” on page 368

awStringAppendByte

```
void awStringAppendByte(  
    BrokerString st,  
    char number);
```

st The destination string.
number The number you are appending to the string.

Appends a string representation of *number* to the end of the destination string *st*.

See also:

“awStringAppend” on page 363

awStringAppendChar

```
void awStringAppendChar(  
    BrokerString st,  
    char c);
```

st The destination string.
c The source character.

Appends the source character *c* to the end of the destination string *st*.

See also:

“awStringAppend” on page 363

awStringAppendDouble

```
void awStringAppendDouble(  
    BrokerString st,  
    double number);
```

st The destination string.
number The double you are appending.

Appends a string representation of *number* to the end of the destination string *st*. The C language locale is used to format the double.

See also:

“awStringAppend” on page 363

“awStringAppendDoubleLocalized” on page 365

awStringAppendDoubleLocalized

```
void awStringAppendDoubleLocalized(  
    BrokerString st,  
    double number);
```

st The destination string.

number The double you are appending.

Appends a string representation of *number* to the end of the destination string *st*. The current locale is used to format the double.

See also:

“awStringAppend” on page 363

“awStringAppendDouble” on page 364

awStringAppendFloat

```
void awStringAppendFloat(  
    BrokerString st,  
    float number);
```

st The destination string.

number The float you are appending.

Appends a string representation of *number* to the end of the destination string *st*. The C language locale is used to format the float.

See also:

“awStringAppend” on page 363

“awStringAppendFloatLocalized” on page 365

awStringAppendFloatLocalized

```
void awStringAppendFloatLocalized(  
    BrokerString st,  
    float number);
```

st The destination string.

number The float you are appending.

Appends a string representation of *number* to the end of the destination string *st*. The current locale is used to format the float.

See also:

“awStringAppend” on page 363

“awStringAppendFloat” on page 365

awStringAppendInteger

```
void awStringAppendInteger(  
    BrokerString st,  
    int number);
```

st The destination string.

number The int you are appending to the string.

Appends a string representation of *number* to the end of the destination string *st*.

See also:

“awStringAppend” on page 363

awStringAppendLong

```
void awStringAppendLong(  
    BrokerString st,  
    BrokerLong number);
```

st The destination string.

number The int you are appending.

Appends a string representation of *number* to the end of the destination string *st*.

See also:

“awStringAppend” on page 363

awStringAppendNString

```
void awStringAppendNString(  
    BrokerString st,  
    char *append,  
    int length);
```

<i>st</i>	The destination string.
<i>append</i>	The source string.
<i>length</i>	The number of bytes of <i>append</i> you want to append to <i>st</i> .

Appends *length* bytes of the source string *append* to the end of the destination string *st*.

See also:

“awStringAppend” on page 363

awStringAppendNUCString

```
void awStringAppendNUCString(
    BrokerString st,
    charUC *append,
    int length);
```

<i>st</i>	The destination string.
<i>append</i>	The Unicode source string.
<i>length</i>	The number of characters to append

Appends the specified number of characters from the Unicode source string *append* to the end of the destination string *st*.

Note:

Any Unicode characters that cannot be represented in ANSI format will be converted UTF-8 format in the destination string.

See also:

“awStringAppend” on page 363

“awStringAppendUCChar” on page 368

“awStringAppendUCString” on page 368

awStringAppendShort

```
void awStringAppendShort(
    BrokerString st,
    short number);
```

<i>st</i>	The destination string.
<i>number</i>	The short you are appending to the string.

Appends a string representation of *number* to the end of the destination string *st*.

See also:

“awStringAppend” on page 363

awStringAppendUCChar

```
void awStringAppendUC(  
    BrokerString st,  
    charUC c);
```

st The destination string.

c The Unicode source character.

Appends the Unicode source character *c* to the end of the destination string *st*.

Note:

Any Unicode character that cannot be represented in ANSI format will be converted UTF-8 format in the destination string.

See also:

“awStringAppend” on page 363

“awStringAppendNUCString” on page 367

“awStringAppendUCString” on page 368

awStringAppendUCString

```
void awStringAppendUCString(  
    BrokerString st,  
    charUC *append);
```

st The destination string.

append The Unicode source string.

Appends the specified Unicode source string *append* to the end of the destination string *st*.

Note:

Any Unicode characters that cannot be represented in ANSI format will be converted UTF-8 format in the destination string.

See also:

“awStringAppend” on page 363

“awStringAppendNUCString” on page 367

“awStringAppendUCChar” on page 368

awStringClip

```
void awStringClip(  
    BrokerString st,  
    int max_length);
```

st The string to be truncated.

max_length The maximum length of the string.

Truncates the string *st* to the specified *max_length*. If *st* already has a length less than or equal to *max_length*, this function has no effect.

awStringDecIndent

```
void awStringDecIndent(  
    BrokerString st);
```

st The string whose indentation level is being set.

Decrements the string indentation level for *st* by one level. See [awStringIndent](#) for a complete description of the string indentation level.

See also:

“awStringIncIndent” on page 369

“awStringIndent” on page 370

“awStringResetIndent” on page 371

awStringIncIndent

```
void awStringIncIndent(BrokerString st);
```

st The string whose indentation level is being set.

Increases the string indentation level for *st* by one. See [awStringIndent](#) for a complete description of string indentation level.

See also:

“awStringDecIndent” on page 369

“awStringIndent” on page 370

“awStringResetIndent” on page 371

awStringIndent

```
void awStringIndent(BrokerString st);
```

st The string being set.

Adds a new line to the string *st* and then adds a number indentation spaces, based on the current indentation level.

This function may be used with `awStringDecIndent` and `awStringIncIndent` as follows:

```
awStringAppend(st,"Hello");
awStringIncIndent(st);
awStringIndent(st);
awStringAppend(st,"First Indent");
awStringIncIndent(st);
awStringIndent(st);
awStringAppend(st,"Second Indent");
awStringIndent(st);
awStringAppend(st,"Still Here");
awStringDecIndent(st);
awStringDecIndent(st);
awStringIndent(st);
awStringAppend(st,"Back at First Level");
```

The following output would result from this code:

```
Hello
First Indent
Second Indent
Still Here
Back at First Level
```

See also:

“awStringDecIndent” on page 369

“awStringIncIndent” on page 369

“awStringResetIndent” on page 371

awStringLength

```
int awStringLength(BrokerString st);
```

st The string.

Returns the length of the string *st*.

awStringPointer

```
char * awStringPointer(BrokerString st);
```

st The string.

Returns a pointer to the string *st*.

See also:

“awDeleteStringWrapper” on page 169

awStringResetIndent

```
void awStringResetIndent(BrokerString st);
```

st The string.

Resets the string indentation level for *st* to zero. See [awStringIndent](#) for a complete description of string indentation level.

See also:

“awStringDecIndent” on page 369

“awStringIncIndent” on page 369

“awStringIndent” on page 370

awThreaded

awThreadedCallbacks

```
BrokerError awThreadedCallbacks(BrokerBoolean enabled);
```

enabled If set to 1 (true) , callback functions will be enabled; 0 (false) means callbacks are to be disabled.

Enables or disables the use of callbacks on a system thread. Use this function in a multi-threaded environment to request callbacks on an event's thread instead of the main thread. Enabling threaded callbacks in a multi-threaded environment allows you to replace the use of event loops and blocking operations.

When using `awThreadedCallbacks`, you cannot invoke any of the following functions:

- [awDispatch](#)

■ `awMainLoop`

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_BAD_STATE</code>	Another thread has already invoked <code>awDispatch</code> or <code>awMainLoop</code> , or an attempt was made to set the threaded callback state to its current state.
<code>AW_ERROR_NOT_IMPLEMENTED</code>	This application was not linked with the multi-threaded version of the Broker library.

See also:

“`awGetEvent`” on page 226

“`awGetEvents`” on page 230

“`awGetEventsWithAck`” on page 231

“`awDispatch`” on page 186

“`awMainLoop`” on page 284

`awTx`

`awTxAbort`

```
BrokerError awTxAbort(BrokerTxClient txclient);
```

txclient The transactional Broker client that is aborting the transaction.

Aborts the transaction.

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_INVALID_CLIENT</code>	The client is not valid.

In addition, any of the communications errors can occur, but if they do, the local client object is still deleted from local memory.

`awTxBeginTransaction`

```
BrokerError awTxBeginTransaction(
    BrokerTxClient txclient,
    char *external_id,
    BrokerLong *tx_id);
```

<i>txclient</i>	The transactional Broker client that is beginning the transaction.
<i>tx_id</i>	The identifier of the transaction, created with the <code>awMakeTxTransactionId</code> function.

Begins a transaction. This gives you a unique transaction id (long `tx_id`) which is unique/Broker.

In addition, any of the communications errors can occur. If an error occurs, the local client object is still deleted from local memory.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The transactional client is not valid.

awTxCanPublish

```
BrokerError awTxCanPublish(
    BrokerTxClient txclient,
    char event_type_name,
    BrokerBoolean can_publish);
```

<i>txclient</i>	The Broker client whose ability to publish is to be tested.
<i>event_type_name</i>	The event type name that the Broker client wishes to publish.
<i>can_publish</i>	A indication of whether or not the Broker client is permitted to publish the specified event type. Set to 1 (true) if the Broker client is permitted; otherwise, set to 0 (false). This parameter is used for output.

Test if you can transactionally publish or deliver an event of a given type. Determines whether or not the specified Broker client can transactionally publish or deliver the specified event type. Upon return, the `can_publish` parameter is set to 1 (true) or 0 (false). If set to 1 (true), the Broker client can publish or deliver events of this event type. Conversely, if set to 0 (false), the Broker client cannot publish or deliver such events.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The transactional client is not valid.
AW_ERROR_NULL_PARAM	The <i>event_type_name</i> or <i>can_publish</i> are NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

awTxCanSubscribe

```
BrokerError awTxCanSubscribe(
    BrokerTxClient txclient,
    char *event_type_name,
    BrokerBoolean *can_subscribe);
```

<i>txclient</i>	The Broker client whose ability to subscribe is to be tested.
<i>event_type_name</i>	The event type name to which the Broker client wishes to subscribe.
<i>can_subscribe</i>	An indication of whether or not the Broker client is permitted to subscribe to the specified event type. Set to 1 (true) if the Broker client is permitted; otherwise, set to 0 (false). This parameter is used for output.

Determines whether or not the specified *txclient* can subscribe to the specified event type. Also determines if an event of the specified type can be delivered to the *txclient*. Upon return, the *can_subscribe* parameter is set to 1 (true) if the Broker client is allowed to subscribe to the event; otherwise *can_subscribe* is set to 0 (false).

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The parameter <i>event_type_name</i> or <i>can_subscribe</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awTxCanPublish” on page 373

“awNewSubscription” on page 301

“awNewSubscriptionFromStruct” on page 302

“awNewSubscriptionsFromStructs” on page 303

“awNewSubscriptionWithId” on page 305

awTxClientToString

```
char * AWCALL awTxClientToString(BrokerTxClient txclient);
```

Get a string with the transactional client information in a form suitable for human viewing. The caller is responsible for freeing the returned value.

awTxCommit

```
BrokerError awTxCommit(
    BrokerTxClient txclient,
    BrokerClient client);
```

Commits the transaction.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The client is not valid.

In addition, any of the communications errors can occur, but if they do, the local client object is still deleted from local memory.

awTxDeliverAckReplyEvent

```
BrokerError awTxDeliverAckReplyEvent(
    BrokerTxClient txclient,
    BrokerEvent request_event,
    BrokerLong publish_seqn);
```

<i>txclient</i>	The Broker client that is transactionally delivering the event.
<i>request_event</i>	The original request event. Used to determine the client ID of the sender of the original request.
<i>publish_seqn</i>	The publish sequence number to use on the reply event. Should be set to zero if you are not using publish sequence numbers.

Delivers an `Adapter::ack` event to the originator of the specified *request_event*. This function properly sets the tag envelope field to match that of the request.

If the `trackId` envelope field was set on *request_event*, that value is copied to the `Adapter::ack` event. If the *request_event* `trackId` envelope field was not set, the `pubId` envelope field from the *request_event* will be used instead.

The Broker client that is to receive the delivered event is not required to have registered a subscription for the event type, but its client group must allow the Broker client to receive the event type.

Note:

An error *will not* be returned if the recipient, represented by the `pubId` field in *request_event*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <i>request_event</i> is invalid.
AW_ERROR_NO_PERMISSION	The <i>client</i> does not have permission to publish <code>Adapter::ack</code> .

See also:

“awTxDeliverErrorReplyEvent” on page 376

“awTxDeliverEvents” on page 378

“awTxDeliverNullReplyEvent” on page 381

“awTxDeliverPartialReplyEvents” on page 382

“awTxDeliverReplyEvent” on page 384

“awTxDeliverReplyEvents” on page 385

awTxDeliverErrorReplyEvent

```
BrokerError awTxDeliverErrorReplyEvent(
    BrokerTxClient txclient,
    BrokerEvent request_event,
    BrokerEvent error_event);
```

<i>txclient</i>	The Broker client that is transactionally delivering the event.
<i>request_event</i>	The original request event. Used to determine the client ID of the sender of the original request.
<i>error_event</i>	The error event to be delivered.

Sends a single error event to the Broker to be transactionally delivered to the Broker client that published the *request_event*. This function properly sets the `tag`, `appSeqn`, and `appLastSeqn` envelope fields.

If the `trackId` envelope field was set on *request_event*, that value is copied to the *error_event* event. If the *request_event* `trackId` envelope field was not set, the `pubId` envelope field from the *request_event* will be used instead.

The *error_event* will be delivered to the Broker client with the client ID contained in the `errorsTo` envelope field of *request_event*, if it was set by the requestor.

Note:

An error *will not* be returned if the recipient, represented by the `pubId` field in *request_event*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>txclient</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <i>request_event</i> or <i>error_event</i> is invalid, the <i>request_event</i> was not received from the Broker, or the <i>error_event</i> does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to publish the <i>error_event</i> .

Possible BrokerError major codes	Meaning
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for <i>error_event</i> does not exist on the Broker.

See also:

“awTxDeliverAckReplyEvent” on page 375

“awTxDeliverEvents” on page 378

“awTxDeliverNullReplyEvent” on page 381

“awTxDeliverPartialReplyEvents” on page 382

“awTxDeliverReplyEvent” on page 384

“awTxDeliverReplyEvents” on page 385

awTxDeliverEvent

```
BrokerError awTxDeliverEvent(
    BrokerTxClient txclient,
    char *dest_id,
    BrokerEvent event);
```

<i>txclient</i>	The Broker client that is transactionally delivering the event.
<i>dest_id</i>	Identifies the Broker client to which the event is to be delivered.
<i>event</i>	The event that is to be delivered.

Transactionally sends the specified *event* to the Broker client with the client identifier represented by *dest_id*. The event is sent to the Broker which, in turn, forwards it to the destination Broker client. The *txclient* that is to receive the delivered event is not required to have registered a subscription for the event type, but its client group must allow the Broker client to receive the event type.

A typical use of this function is when your Broker client replies to a request event from another Broker client. In such a case, you can obtain the *dest_id* by extracting it from the envelope of the request event, as described on [“Delivering Events” on page 72](#).

Note:

An error *will not* be returned if the recipient, represented by *dest_id*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>client</i> parameter, has been destroyed or disconnected.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT_ID	The destination client ID contains illegal characters.
AW_ERROR_INVALID_EVENT	The <i>event</i> is not valid.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to publish the event type.
AW_ERROR_NULL_PARAM	The parameter <i>dest_id</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for the event does not exist on the Broker.

See also:

“awTxDeliverAckReplyEvent” on page 375

“awTxDeliverEvents” on page 378

“awTxDeliverNullReplyEvent” on page 381

“awTxDeliverPartialReplyEvents” on page 382

“awTxDeliverReplyEvent” on page 384

“awTxPublishEvent” on page 390

“awTxPublishEvents” on page 391

awTxDeliverEvents

```
BrokerError awTxDeliverEvents(
    BrokerTxClient txclient,
    char *dest_id,
    int n,
    BrokerEvent *events);
```

<i>txclient</i>	The Broker client that is transactionally sending the events.
<i>dest_id</i>	The identifier of the Broker client to which the events are to be delivered.
<i>n</i>	The number of events in the array.
<i>events</i>	The array of events that are to be delivered.

Deliver multiple events. Gives an array of events to the Broker to have them all delivered to the *txclient* with the given client ID. Either all of the events or none of them are delivered. *n* is the number of events in the events array. No error is returned if there is no client using the destination client ID.

Note:

An error *will not* be returned if the recipient, represented by *dest_id*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>txclient</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_CLIENT_ID	The destination client ID contains illegal characters.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to publish the event type.
AW_ERROR_NULL_PARAM	The parameter <i>dest_id</i> or <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awTxDeliverAckReplyEvent” on page 375

“awTxDeliverErrorReplyEvent” on page 376

“awTxDeliverEvent” on page 377

“awTxDeliverEventsWithAck” on page 379

“awTxDeliverNullReplyEvent” on page 381

“awTxDeliverPartialReplyEvents” on page 382

“awTxDeliverReplyEvent” on page 384

“awTxDeliverEvents” on page 378

“awTxPublishEvent” on page 390

“awTxPublishEvents” on page 391

“awTxPublishEventsWithAck” on page 392

awTxDeliverEventsWithAck

```
BrokerError awTxDeliverEventsWithAck(
    BrokerTxClient txclient,
    char *dest_id,
    int n,
    BrokerEvent *events,
    int ack_type,
    int n_acks,
    BrokerLong *ack_seqn);
```

<i>txclient</i>	The Broker client that wishes to transactionally publish the events.
<i>dest_id</i>	The identifier of the Broker client to which the events are to be delivered.
<i>n</i>	The number of events in the events array.
<i>events</i>	The array of events to be delivered.
<i>ack_type</i>	Determines how the events will be acknowledged and must be set to one of the following: <ul style="list-style-type: none"> ■ AW_ACK_NONE ■ AW_ACK_AUTOMATIC ■ AW_ACK_THROUGH ■ AW_ACK_SELECTIVE
<i>n_acks</i>	The number of sequence numbers in <i>ack_seqn</i> .
<i>ack_seqn</i>	The array of sequence numbers to be acknowledged if <i>ack_type</i> is set to AW_ACK_THROUGH or AW_ACK_SELECTIVE.

Sends the array of *events* to the Broker for delivery to the Broker client destination specified by *dest_id*. You also have one of several options for acknowledging events already received by this client. Either all events or none will be delivered.

Note:

An error *will not* be returned if the recipient, represented by *dest_id*, no longer exists.

The setting of the *ack_type* and *ack_seqn* parameters will determine which events received by this client are to be acknowledged.

ack_type	ack_seqn	Result
AW_ACK_NONE	Not applicable.	No events are acknowledged.
AW_ACK_AUTOMATIC	Not applicable.	All events received by the client are acknowledged.
AW_ACK_THROUGH	<i>ack_seqn</i> [0] contains the sequence number of the last event to be acknowledged. If set to 0, the behavior will be the same as AW_ACK_AUTOMATIC	Acknowledges all events up to and including the sequence number specified by <i>ack_seqn</i> [0]. If the <i>n_acks</i> argument is zero, no events will be acknowledged.
AW_ACK_SELECTIVE	<i>ack_seqn</i> contains the sequence numbers of	Acknowledges the specific events whose sequence number are contained in <i>ack_seqn</i> . The <i>n_acks</i> argument must specify the number of

ack_type	ack_seqn	Result
	the specific events to be acknowledged.	sequence numbers contained in <i>ack_seqn</i> . All sequence numbers must be greater than zero.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_ACKNOWLEDGEMENT	The parameter <i>ack_seqn</i> contained an invalid sequence number. The events were not sent to the Broker.
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> is not valid.
AW_ERROR_INVALID_CLIENT_ID	The <i>dest_id</i> contains invalid characters.
AW_ERROR_INVALID_EVENT	One of the events contained in <i>events</i> is invalid.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to publish all of the event types.
AW_ERROR_NULL_PARAM	The parameter <i>events</i> or <i>dest_id</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awTxCanPublish” on page 373

“awTxDeliverErrorReplyEvent” on page 376

“awTxDeliverEvents” on page 378

“awTxPublishEvent” on page 390

“awTxPublishEvents” on page 391

“awTxPublishEventsWithAck” on page 392

awTxDeliverNullReplyEvent

```
BrokerError awTxDeliverNullReplyEvent(
    BrokerTxClient txclient,
    BrokerEvent request_event,
    char *reply_event_type_name,
    BrokerLong publish_seqn);
```

txclient The Broker client that is transactionally delivering the reply event.

request_event The original request event. Used to determine the client ID of the sender of the original request.

<i>reply_event_type_name</i>	The type name of the null reply event.
<i>publish_seqn</i>	The publish sequence number for the reply event. Set to zero if your application is not using publish sequence numbers.

Transactionally delivers a null event of type *reply_event_type_name* to the publisher of the *request_event*. The envelope tag, appSeqn, and appLastSeqn fields are set to indicate that this is a null event. This indicates that the request was successful and resulted in no data.

If the trackId envelope field was set on *request_event*, that value is copied to the null reply event. If the *request_event* trackId envelope field was not set, the pubId envelope field from the *request_event* will be used instead.

Note:

An error *will not* be returned if the recipient, represented by the pubId field in *request_event*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>txclient</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <i>request_event</i> is invalid.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to publish the event type.
AW_ERROR_NULL_PARAM	The parameter <i>reply_event_type_name</i> is NULL.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for the reply event does not exist on the Broker.

See also:

“awTxDeliverAckReplyEvent” on page 375

“awTxDeliverErrorReplyEvent” on page 376

“awTxDeliverEvents” on page 378

“awTxDeliverPartialReplyEvents” on page 382

“awTxDeliverReplyEvent” on page 384

“awTxDeliverReplyEvents” on page 385

awTxDeliverPartialReplyEvents

```
BrokerError awTxDeliverPartialReplyEvents(
    BrokerTxClient txclient,
    BrokerEvent request_event,
```

```
int n,
BrokerEvent *events,
int flag,
int *reply_token);
```

<i>txclient</i>	The Broker client that is transactionally delivering the events.
<i>request_event</i>	The original request event. Used to determine the client ID of the sender of the original request.
<i>n</i>	The number of events in the reply.
<i>events</i>	The array of reply events that are to be delivered.
<i>flag</i>	Indicates if there are more events to be sent as part of this reply. Must be one of the following: <ul style="list-style-type: none"> ■ AW_REPLY_FLAG_START ■ AW_REPLY_FLAG_CONTINUE ■ AW_REPLY_FLAG_START_AND_END ■ AW_REPLY_FLAG_END
<i>reply_token</i>	The address of a <code>int</code> that can be used by the function as temporary storage between calls.

Delivers an *event* array of size *n* to the Broker to be transactionally delivered to the Broker client that originally published *request_event*. No error will be returned if the Broker client using that ID no longer exists. Either all of the events or none of them will be delivered. This function properly sets the `tag`, `appSeqn`, and `appLastSeqn` envelope fields.

If the `trackId` envelope field was set on *request_event*, that value is copied to the reply events. If the *request_event* `trackId` envelope field was not set, the `pubId` envelope field from the *request_event* will be used instead.

This function is used to deliver parts of a set of replies in groups. When called the first time, *flag* should be `AW_REPLY_FLAG_START`. After doing this, additional calls can be made with other *flag* values. During intermediate replies, *flag* should be `AW_REPLY_FLAG_CONTINUE`. On the final call, *flag* should be `AW_REPLY_FLAG_END`. It is important that the ending call be made with *n* is set to at least 1.

Calling this function with *flag* set to `AW_REPLY_FLAG_START_AND_END` allows the entire result to be passed to this function in one call.

The *reply_token* value will be set and modified by this function during calls. It exists to carry information between calls and has no meaning to the caller.

Note:

An error *will not* be returned if the recipient, represented by the `pubId` field in *request_event*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>txclient</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <i>request_event</i> or one of the reply <i>events</i> is invalid, <i>request_event</i> was not received from the Broker, or one the reply <i>events</i> does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to publish the <i>reply_event</i> type.
AW_ERROR_NULL_PARAM	The parameter <i>events</i> or <i>reply_token</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero or <i>flag</i> does not contain a valid value.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for one of the reply <i>events</i> does not exist on the Broker.

See also:

“awTxDeliverAckReplyEvent” on page 375

“awTxDeliverErrorReplyEvent” on page 376

“awTxDeliverEvents” on page 378

“awTxDeliverNullReplyEvent” on page 381

“awTxDeliverReplyEvent” on page 384

“awTxDeliverReplyEvents” on page 385

awTxDeliverReplyEvent

```
BrokerError awTxDeliverReplyEvent( BrokerTxClient txclient,
BrokerEvent request_event, BrokerEvent event);
```

txclient The Broker client that is transactionally sending the event.

request_event The original request event for which this reply is to be delivered.

event The reply event to be delivered.

Transfers the *event* to the Broker to be transactionally delivered to the Broker client who sent the original *request_event*. No error is returned if the Broker client with the destination identifier specified in *request_event* no longer exists. This function properly sets the *tag*, *appSeqn*, and *appLastSeqn* envelope fields on *event*.

If the `trackId` envelope field was set on `request_event`, that value is copied to the reply event. If the `request_event` `trackId` envelope field was not set, the `pubId` envelope field from the `request_event` will be used instead.

Note:

An error *will not* be returned if the recipient, represented by the `pubId` field in `request_event`, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <code>txclient</code> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <code>request_event</code> or the <code>reply_event</code> is invalid, <code>request_event</code> was not received from the Broker, or the <code>reply_event</code> does not match its type definition.
AW_ERROR_NO_PERMISSION	The <code>txclient</code> does not have permission to publish the <code>reply_event</code> type.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for <code>event</code> does not exist on the Broker.

See also:

“`awTxDeliverErrorReplyEvent`” on page 376

“`awTxDeliverEvents`” on page 378

“`awTxDeliverAckReplyEvent`” on page 375

“`awTxDeliverNullReplyEvent`” on page 381

“`awTxDeliverReplyEvents`” on page 385

awTxDeliverReplyEvents

```
BrokerError awTxDeliverReplyEvents(
    BrokerTxClient txclient,
    BrokerEvent request_event,
    int n,
    BrokerEvent *events);
```

<code>txclient</code>	The Broker client that is transactionally sending the event.
<code>request_event</code>	The original request event for which this reply is to be delivered.
<code>n</code>	The number of reply events to be delivered.
<code>events</code>	An array of reply event to be delivered.

Transfers the multiple *events* to the Broker to be delivered transactionally to the Broker client who sent the original *request_event*. This function properly sets the *tag*, *appSeqn*, and *appLastSeqn* envelope fields on all the events contained in the *events* array.

If the *trackId* envelope field was set on *request_event*, that value is copied to the reply events. If the *request_event* *trackId* envelope field was not set, the *pubId* envelope field from the *request_event* will be used instead.

Note:

An error *will not* be returned if the recipient, represented by the *pubId* field in *request_event*, no longer exists.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The Broker client, represented by the <i>txclient</i> parameter, has been destroyed or disconnected.
AW_ERROR_INVALID_EVENT	The <i>request_event</i> or one of the reply <i>events</i> is invalid, <i>request_event</i> was not received from the Broker, or one of the reply <i>events</i> does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to publish the <i>reply_event</i> type.
AW_ERROR_NULL_PARAM	The parameter <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type for <i>event</i> does not exist on the Broker.

See also:

“awTxDeliverErrorReplyEvent” on page 376

“awTxDeliverEvents” on page 378

“awTxDeliverAckReplyEvent” on page 375

“awTxDeliverNullReplyEvent” on page 381

“awTxDeliverReplyEvent” on page 384

awTxGetEvent

```
BrokerError awTxGetEvent(
    BrokerTxClient txclient,
    int msec,
    BrokerEvent *event);
```

txclient The Broker client whose next event is to be obtained.

<i>msecs</i>	Number of milliseconds to wait for an event before timing out. If set to <code>AW_INFINITE</code> , this function will wait indefinitely.
<i>event</i>	The event that is returned. This parameter is used for output.

Acknowledges all previously retrieved events for *txclient*, then obtains a single event for *txclient*, if available. If no events are currently available, this function will wait for the number of milliseconds specified by *msecs*. If the wait time expires, this function returns with error code `AW_ERROR_TIMEOUT`. Any *event* that is obtained may be one for which the *client* has registered a subscription, or it may be a delivered event.

Note:

Before exiting, Broker clients with an explicit-destroy life cycle that are using `awGetEvent` should explicitly acknowledge the receipt sequence number of the last event received using either [awAcknowledge](#) or [awAcknowledgeThrough](#). Failure to do so will result in the last event being received again the next time you connect the Broker client.

Using this function on a client that has registered callback functions will temporarily disable the callback mechanism for this Broker client until this function returns. The caller is responsible for calling `awDeleteEvent` for the output event.

Note:

The Broker will delete all guaranteed events from the event queue once they are acknowledged. If you wish to receive an event without acknowledging any previously retrieved events, use the [awTxGetEventsWithAck](#) function and specify a sequence number of -1.

Possible BrokerError major codes	Meaning
<code>AW_ERROR_INTERRUPTED</code>	The awInterruptGetEvents function was invoked.
<code>AW_ERROR_INVALID_CLIENT</code>	The <i>txclient</i> has been destroyed or disconnected.
<code>AW_ERROR_NULL_PARAM</code>	The parameter <i>events</i> is NULL.
<code>AW_ERROR_OUT_OF_RANGE</code>	The <i>msecs</i> parameter is less than -1.
<code>AW_ERROR_TIMEOUT</code>	The <i>msecs</i> time-out interval expired before an event arrived.

See also:

- “awDispatch” on page 186
- “awGetEvents” on page 230
- “awGetEventsWithAck” on page 231
- “awInterruptGetEvents” on page 279
- “awMainLoop” on page 284

“awThreadedCallbacks” on page 371

awTxGetEvents

```
BrokerError awTxGetEvents(
    BrokerTxClient txclient,
    int max_events,
    int msec,
    int *n,
    BrokerEvent **events);
```

<i>txclient</i>	The Broker client requesting the event.
<i>max_events</i>	The maximum number of events to be returned.
<i>msecs</i>	The number of milliseconds to wait for the events before timing out. If set to <code>AW_INFINITE</code> , this function will wait indefinitely.
<i>n</i>	The number of events returned. This parameter is used for output.
<i>events</i>	An array of events. This parameter is used for output.

Acknowledges all previously retrieved events for *txclient*, then obtains one or more events for *txclient*, if available. If no events are currently available, this function will wait for the number of milliseconds specified by *msecs*. Any *event* that is obtained may be one for which the *txclient* has registered a subscription, or it may be a delivered event.

If the wait time expires, this function returns a `BrokerError` with a major code of `AW_ERROR_TIMEOUT` and *n* set to zero.

Note:

Before exiting, Broker clients with an explicit-destroy life cycle that are using `awTxGetEvents` should explicitly acknowledge the receipt sequence number of the last event received using either `awAcknowledge` or `awAcknowledgeThrough`. Failure to do so will result in the last event being received again the next time you connect the Broker client.

Using this function on a client that has registered callback functions will temporarily disable the callback mechanism for that client until this function returns.

The caller is responsible for calling `awDeleteEvent` on each event and then calling `free` on the array itself.

Note:

The Broker will delete all guaranteed events from the event queue once they are acknowledged. If you wish to receive multiple events without acknowledging any previously retrieved events, use the `awGetEventsWithAck` function and specify a sequence number of -1.

Possible BrokerError major codes

Meaning

`AW_ERROR_INTERRUPTED`

The `awInterruptGetEvents` function was invoked.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> has been destroyed or disconnected.
AW_ERROR_NULL_PARAM	The <i>n</i> or <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The <i>msecs</i> parameter is less than -1 or <i>max_events</i> is less than zero.
AW_ERROR_TIMEOUT	The <i>msecs</i> time-out interval expired before any events arrived.

See also:

“awDispatch” on page 186

“awGetEvent” on page 226

“awGetEventsWithAck” on page 231

“awInterruptGetEvents” on page 279

“awMainLoop” on page 284

“awThreadedCallbacks” on page 371

awTxGetEventsWithAck

```
BrokerError awTxGetEventsWithAck(
    BrokerTxClient txclient,
    int max_events,
    BrokerLong seqn,
    int msecs,
    int *n,
    BrokerEvent **events);
```

<i>txclient</i>	The Broker client requesting the events.
<i>max_events</i>	The maximum number of events to be returned.
<i>seqn</i>	Specifies the last event to acknowledge. If set to zero, all previously received events that have not been acknowledged will be acknowledged. If set to -1, no acknowledgment is done at all.
<i>msecs</i>	The number of milliseconds to wait for the events before timing out. If set to AW_INFINITE, this function will wait indefinitely.
<i>n</i>	The number of events returned. This parameter is used for output.
<i>events</i>	An array of events. This parameter is used for output.

Acknowledges all the events received by *txclient*, up to the event specified by *seqn* and then obtains one or more events. If *seqn* is set to -1, no previously retrieved events will be acknowledged.

The *events* that are obtained may be those for which the *client* has registered a subscription, they may be delivered events, or both.

Calling `awTxGetEventsWithAck` on a client that has registered callback functions will temporarily disable the callback mechanism for that client until this function returns. For more information on acknowledging events see [“Using Sequence Numbers” on page 397](#). If the wait time expires, this function returns a `BrokerError` with a major code of `AW_ERROR_TIMEOUT` and *n* set to zero.

The caller is responsible for calling `awDeleteEvent` on each event and then for calling `free` on the array itself.

Note:

The Broker will delete all guaranteed events from the event queue once they are acknowledged.

Possible <code>BrokerError</code> major codes	Meaning
<code>AW_ERROR_INTERRUPTED</code>	The <code>awInterruptGetEvents</code> function was invoked.
<code>AW_ERROR_INVALID_CLIENT</code>	The <i>client</i> has been destroyed or disconnected.
<code>AW_ERROR_NULL_PARAM</code>	The <i>n</i> or <i>events</i> is <code>NULL</code> .
<code>AW_ERROR_OUT_OF_RANGE</code>	The <i>msecs</i> parameter is less than -1 or <i>max_events</i> is less than zero.
<code>AW_ERROR_TIMEOUT</code>	The <i>msecs</i> time-out interval expired before any events arrived.

See also:

“`awDispatch`” on page 186

“`awGetEvent`” on page 226

“`awGetEvents`” on page 230

“`awInterruptGetEvents`” on page 279

“`awMainLoop`” on page 284

“`awThreadedCallbacks`” on page 371

awTxPublishEvent

```
BrokerError awTxPublishEvent(
    BrokerTxClient txclient,
    BrokerEvent event);
```

txclient The Broker client to transactionally publish the event.

event The event to be published.

Transactionally publishes the specified event. The event is sent to the Broker then is given to all subscribing clients.

Note:

This function can fail if the Broker client does not have permission to publish the event type, based on its client group, or if the event is not properly formed. See [awTxCanPublish](#) for more information.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> is not valid.
AW_ERROR_INVALID_EVENT	The <i>event</i> is not valid or the event does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to publish the event type.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

awTxPublishEvents

```
BrokerError awTxPublishEvent(
    BrokerTxClient txclient,
    int n,
    BrokerEvent *event);
```

txclient The Broker client to transactionally publish the event.

n The number of events in the array.

event The event to be published.

Transactionally publishes multiple events. Gives an array of events to the to have them all delivered to the client with the given client ID. Either all of the events or none of them are delivered. *n* is the number of events in the events array.

Note:

This function can fail if the Broker client does not have permission to publish the event type, based on its client group, or if the event is not properly formed. See [awTxCanPublish](#) for more information.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> is not valid.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_EVENT	The <i>event</i> is not valid or the event does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to publish all of the event type.
AW_ERROR_NULL_PARAM	The parameter <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

awTxPublishEventsWithAck

```
BrokerError awTxPublishEventsWithAck(
    BrokerTxClient txclient,
    int n,
    BrokerEvent *events,
    int ack_type,
    int n_acks,
    BrokerLong *ack_seqn);
```

<i>txclient</i>	The Broker client to transactionally publish the events.
<i>n</i>	The number of events in the events array.
<i>events</i>	The array of events to be published.
<i>ack_type</i>	Determines how the events will be acknowledged and must be set to one of the following: <ul style="list-style-type: none"> ■ AW_ACK_NONE ■ AW_ACK_AUTOMATIC ■ AW_ACK_THROUGH ■ AW_ACK_SELECTIVE
<i>n_acks</i>	The number of sequence numbers in <i>ack_seqn</i> .
<i>ack_seqn</i>	The array of sequence numbers to be acknowledged if <i>ack_type</i> is set to AW_ACK_THROUGH or AW_ACK_SELECTIVE.

Sends the array of *events* to the Broker for publication with one of several options for acknowledging events already received by this *txclient*. Either all events or none will be published.

Note:

This function can fail if the Broker client does not have permission to publish this event type, based on its client group, or if the event is not properly formed. See [awTxCanPublish](#) for more information.

The setting of the *ack_type* and *ack_seqn* parameters will determine which events received by this client are to be acknowledged.

ack_type	ack_seqn	Result
AW_ACK_NONE	Not applicable.	No events are acknowledged.
AW_ACK_AUTOMATIC	Not applicable.	All events received by the client are acknowledged.
AW_ACK_THROUGH	<i>ack_seqn[0]</i> contains the sequence number of the last event to be acknowledged. If set to 0, the behavior will be the same as AW_ACK_AUTOMATIC	Acknowledges all events up to and including the sequence number specified by <i>ack_seqn[0]</i> . If the <i>n_acks</i> argument is zero, no events will be acknowledged.
AW_ACK_SELECTIVE	<i>ack_seqn</i> contains the sequence numbers of the specific events to be acknowledged.	Acknowledges the specific events whose sequence numbers are contained in <i>ack_seqn</i> . The <i>n_acks</i> parameter must specify the number of sequence numbers contained in <i>ack_seqn</i> . All sequence numbers must be greater than zero.

Possible BrokerError major codes	Meaning
AW_ERROR_INVALID_ACKNOWLEDGEMENT	The parameter <i>ack_seqn</i> contained an invalid sequence number. The events were not sent to the Broker.
AW_ERROR_INVALID_CLIENT	The <i>txclient</i> is not valid.
AW_ERROR_INVALID_EVENT	One of the events in <i>events</i> is not valid or the event does not match its type definition.
AW_ERROR_NO_PERMISSION	The <i>txclient</i> does not have permission to publish all of the event types.
AW_ERROR_NULL_PARAM	The parameter <i>events</i> is NULL.
AW_ERROR_OUT_OF_RANGE	The parameter <i>n</i> is less than zero.
AW_ERROR_UNKNOWN_EVENT_TYPE	The event type does not exist on the Broker.

See also:

“awTxCanPublish” on page 373

“awTxDeliverErrorReplyEvent” on page 376

“awTxDeliverEvents” on page 378

“awTxDeliverEventsWithAck” on page 379

“awTxPublishEvent” on page 390

“awTxPublishEvents” on page 391

awType

awTypeDefToString

```
char * awTypeDefToString(  
    BrokerTypeDef type_def);
```

type_def The type definition whose string representation is to be returned.

Returns a string representing the Broker type definition, *type_def*. Returns NULL if *type_def* is invalid. The caller is responsible for calling `free` on the return value.

awUnlock

awUnlockTypeDefCache

```
void awUnlockTypeDefCache();
```

Unlocks your application's event type definition cache. See [“Event Type Definition Cache” on page 117](#) for more information on using this function.

See also:

“awLockTypeDefCache” on page 284

“awFlushTypeDefCache” on page 198

awValidate

awValidateEvent

```
BrokerError awValidateEvent(  
    BrokerClient client,  
    BrokerEvent event);
```

client The Broker client whose context is to be used to validate the event.

event The event to be validated.

Validates the *event* in the context of the specified *client*. This function checks if the *event* would be valid for a client other than the one for which it was created. An event passes validation if its event type exists on the Broker and if the field names and types match the Broker's definition for the event.

Possible BrokerError major codes	Meaning
AW_ERROR_FIELD_TYPE_MISMATCH	The <i>event</i> did not pass validation.
AW_ERROR_INVALID_CLIENT	The <i>client</i> is invalid or, if <i>client</i> is NULL, the <i>event</i> is not type-checked.
AW_ERROR_INVALID_EVENT	The <i>event</i> is invalid.

15 Using Sequence Numbers

■ Overview	398
■ Sequence Numbers	398
■ Publisher Sequence Numbers	398
■ Receipt Sequence Numbers	399

Overview

This chapter describes how to use event sequence numbers to ensure reliable event delivery. Reading this chapter should help you understand how to:

- Use publish sequence numbers.
- Use receipt sequence numbers.

Sequence Numbers

Sequence numbers in the webMethods Broker are designed to ensure that events are delivered reliably. The use of sequence numbers is optional, but is recommended if your application requires reliable delivery of events and is capable of storing and tracking these numbers.

Sequence numbers can be used between a publishing application and the Broker or between the Broker and a receiving application. Here are some common characteristics of both publish and receipt sequence numbers:

- Sequence numbers are 64-bit unsigned integers that contain non-zero values.
- Sequence numbers are always incremented and are assumed to never wrap back to 1.

Publisher Sequence Numbers

You do not need to use publisher sequence numbers if your only concern is whether or not a published or delivered event gets to the Broker. If your application's publish or deliver function returns successfully, the event was transmitted to the Broker. If the publish or deliver function fails, you may simply send the event again.

When you re-send an event, there is a small chance a duplicate event will be presented to the Broker. The Broker may successfully receive a published event, but then encounter some error that prevents it from notifying your application. Avoiding this situation can be important if your application is dealing with financial transactions, for example.

When your application uses publisher sequence numbers, it allows the Broker to recognize and discard duplicate events. When used properly, this eliminates the window of opportunity for the presentation of duplicate events to the Broker.

Using Publisher Sequence Numbers

When the Broker receives an event with a sequence number that has not been set or is set to zero, it assumes that event sequence numbering rules are not to be applied to the event. If your application does not wish to use publish sequence numbers, it simply should not set them or set them to zero.

Your application may set a non-zero publish sequence number for an event by calling the [awSetEventPublishSequenceNumber](#) function prior to publishing or delivering the event. The [awDeliverAckReplyEvent](#) and [awDeliverNullReplyEvent](#) functions allow you to specify a publish sequence number directly.

When the event is published or delivered, the Broker will discard the event if the sequence number is less than or equal to the highest sequence number received from that Broker client so far.

Note:

To ensure that events are not discarded, publishing applications should use increasing sequence numbers. The sequence number values, however, may be incremented by a value greater than one.

Maintaining Publish Sequence Number State

To make the most of publisher sequence numbers, publishing applications should store, in some reliable way, the sequence numbers of successfully transmitted events. This will allow them to continue publishing where they left off if they should terminate unexpectedly and need to be restarted.

Your application may use the `awGetClientLastPublishSequenceNumber` function to determine the highest sequence number that it has used so far.

Your application can obtain the current setting on an event's sequence number by calling `awGetEventPublishSequenceNumber`.

Receipt Sequence Numbers

Receipt sequence numbers can be used to prevent the Broker from presenting duplicate events to your application's `BrokerClient` when it retrieves the next event. The Broker always sets sequence numbers on the events it presents to receiving clients. If your receiving Broker client acknowledges an event's sequence number, the Broker will not present that event again.

Note:

Volatile events are not assigned sequence numbers by the Broker, so they cannot be specifically acknowledged. Volatile events are deleted from the client's event queue as soon as they are sent to a Broker client.

Default Acknowledgment

The following function calls will automatically acknowledge all previously retrieved events for your receiving Broker client, effectively ignoring sequence numbers:

- `awGetEvent`
- `awGetEvents`

This default behavior avoids the loss of events from the event queue in cases where your application crashes while processing an event, because the event is not acknowledged until the next time your Broker client requests an event. The next time your Broker client requests an event, the Broker will assume the last retrieved event has already been processed.

Your Broker client can acknowledge the received event by either asking for another event, using the same function call, or by explicitly calling `awAcknowledge`. A Broker client that is implicitly

acknowledging events would normally call `awAcknowledge` only if it was planning to exit and wanted to acknowledge all previously received events without actually receiving any more events.

Note:

Before exiting, Broker clients with an *explicit-destroy* life cycle that are using `awGetEvent` or `awGetEvents` with implicit acknowledgment should explicitly acknowledge the receipt sequence number of the last event received using either `awAcknowledge` or `awAcknowledgeThrough`. Failure to do so will result in the last event being received again the next time you connect the Broker client.

Default Acknowledgment With Callbacks

Your Broker client has much less flexibility in acknowledging events when it registers callback functions to process events, as described in “Using the Callback Model” on page 75. All callback functions must either return `1` (`true`) or `0` (`false`). If `true` is returned, the event will automatically be acknowledged. If `false` is returned, the event is assumed to have not been successfully handled and, therefore, it is not acknowledged.

Important:

If an event meets the criteria for more than one callback function and any one of the callback functions returns `false`, that event will not be acknowledged.

Explicitly Acknowledging Events

By explicitly acknowledging events, your application retains a greater degree of control over what events the Broker will maintain.

Your Broker client can use the `awGetEventsWithAck` function to obtain one or more events from the Broker while simultaneously acknowledging an event through a specified sequence number. If the sequence number is set to `-1`, no acknowledgment is sent and you may then use one of the following two functions to explicitly acknowledge the events you choose.

Your Broker client can use the `awAcknowledge` function to acknowledge the receipt of a single event with the specified sequence number. If a sequence number of zero is specified, all previously unacknowledged events will be acknowledged.

Your Broker client can use the `awAcknowledgeThrough` function to acknowledge the receipt of all events up to and including the event with the sequence number specified. If a sequence number of zero is specified, all previously unacknowledged events will be acknowledged.

The `awDeliverEventsWithAck` method can be used by your Broker client to deliver one or more events while, at the same time, acknowledging the receipt of one or more events.

The `awPublishEventsWithAck` method can be used by your Broker client to publish one or more events while, at the same time, acknowledging the receipt of one or more events.

Maintaining Receipt Sequence Number State

To make the most of explicitly acknowledging receipt sequence numbers, your receiving applications should store, in some reliable way, the receipt sequence numbers of successfully processed events. This will allow your applications to continue receiving events where they left off if they should terminate and need to be restarted.

You can use the [awGetEventReceiptSequenceNumber](#) method to obtain the receipt sequence number from a received event.

Other Considerations with Receipt Sequence Numbers

The Broker guarantees that events from a single publishing Broker client will not be processed out of order. This has important implications when more than one Broker client is sharing the same event queue because the Broker will not return an event to Broker client which is incapable of acknowledging the event.

The following table shows an example event queue containing event received from three different publishing Broker clients; Client A, Client B, and Client C.

Publishing Broker client	Event Queue Position
Broker client A	1
Broker client B	2
Broker client A	3
Broker client C	4
Broker client B	5
Broker client C	6

Consider these steps:

1. Broker client X receives the event from queue position 1 without acknowledging the event.
2. Broker client Y receives the event from queue position 2 without acknowledging the event.
3. Broker client Y then asks for another event and is given the event from queue position 4.

Since the last event from publishing Broker client A has not yet been acknowledged, the next event in the queue from Broker client A cannot be given to a different receiving client.

By enforcing these acknowledgment rules, the Broker allows you to write client applications that process events from a single queue across multiple threads of control.

A Error Definitions

This appendix contains the major error codes that you might encounter when using the C Library.

You can use the [awErrorToString](#) method to obtain a character string that briefly describes the error associated with a particular `BrokerError`.

The [awErrorToCompleteString](#) method lets you obtain a character string that specifically describes the error associated with a particular `BrokerError`.

AW_ERROR_BAD_STATE

You invoked an API function that conflicts with the current system state. For example:

- You attempted to register a callback for a subscription ID before registering a general callback.
- You attempted to invoke the `awDispatch` function within a callback function.

AW_ERROR_BROKER_FAILURE

An unexpected failure occurred while the Broker was processing your request. This error can have two possible meanings:

- The API could not correctly process an error into one of the other errors.
- A Broker failure has occurred, such as running out of memory or a corrupted data store.

AW_ERROR_BROKER_NOT_RUNNING

While attempting to create or reconnect a Broker client, the specified host was found but no Broker was running on that host.

AW_ERROR_CLIENT_CONTENTION

On reconnect Broker client calls, the client is either already in use, or the client has shared state and the maximum has already been reached.

AW_ERROR_CLIENT_EXISTS

The client ID specified when creating a new Broker client is already in use.

AW_ERROR_COMM_FAILURE

A generic communications fault has occurred. Network failures cause this error to be returned.

AW_ERROR_CONNECTION_CLOSED

The connection to the Broker was closed before or during the operation you requested.

AW_ERROR_CORRUPT

The data object on which you are operating is corrupt. Currently only detected on `BrokerEvent` objects.

AW_ERROR_FIELD_NOT_FOUND

Attempt to operate on a `BrokerEvent` field that does not exist.

AW_ERROR_FIELD_TYPE_MISMATCH

The specified event field is not of the expected type. For example, using the `awSetStringField` function on an event field of type `int` will generate this error.

AW_ERROR_FILE_NOT_FOUND

The specified file could not be found or could not be opened.

AW_ERROR_FILTER_PARSE

An error occurred while parsing the filter string specified when creating a new filter with the `awNewBrokerFilter` function.

AW_ERROR_FILTER_RUNTIME

A runtime error, such as division by zero, occurred while parsing the filter specified with the `awMatchFilter` function.

AW_ERROR_FORMAT

This can result from some protocol failures. It is mainly issued by the `BrokerString` calls to parse values out of strings.

AW_ERROR_HOST_NOT_FOUND

The host specified with the `awNewBrokerClient` function or the `awReconnectBrokerClient` function could not be located.

AW_ERROR_INCOMPATIBLE_VERSION

The Broker is running an older version of the product than this library.

AW_ERROR_INTERRUPTED

An invocation of `awGetEvent`, `awGetEvents`, `awGetEventsWithAck`, or `awDispatch` was interrupted by the invocation of the `awInterruptGetEvents` or the `awInterruptDispatch` function.

AW_ERROR_INVALID_ACKNOWLEDGEMENT

An attempt was made to acknowledge a sequence number that was out of order or which was not been assigned to your Broker client.

AW_ERROR_INVALID_CLIENT

The `BrokerClient` passed to the function is either disconnected or destroyed.

AW_ERROR_INVALID_CLIENT_ID

The client ID passed to the function contained illegal characters. See [“Parameter Naming Rules” on page 407](#) for details on the proper format of client identifiers.

AW_ERROR_INVALID_DESCRIPTOR

The `BrokerConnectionDescriptor` object passed to the function has been deleted.

AW_ERROR_INVALID_EVENT

The `BrokerEvent` object passed to the function has been deleted.

AW_ERROR_INVALID_EVENT_TYPE_NAME

The event type name contains illegal characters, reserved words, or has components over 255 characters in length. See “[Parameter Naming Rules](#)” on page 407 for details on the proper format of event type names.

AW_ERROR_INVALID_FIELD_NAME

The field name contains illegal characters, reserved words, or has components over 255 characters in length. See “[Parameter Naming Rules](#)” on page 407 for details on the proper format of event type names.

AW_ERROR_INVALID_FILTER

The `BrokerFilter` object passed to the function has been deleted.

AW_ERROR_INVALID_PORT

The specified port number contains invalid characters or is out of range.

AW_ERROR_INVALID_SUBSCRIPTION

The subscription requested with the `awNewSubscription` function contained a null `event_type_name` parameter, a negative subscription ID, or a filter string that could not be parsed.

This error may also be returned if the subscription passed to the `awCancelSubscription` function could not be found by the Broker.

AW_ERROR_INVALID_TYPE

An attempt was made to set an event field, using a function such as `awSetField`, to a value that did not match the field's type.

This error may also be reported if an internal failure is encountered by the API.

AW_ERROR_INVALID_TYPECACHE

An internal error occurred with the event type definition cache. This does not represent a user error.

AW_ERROR_INVALID_TYPEDEF

The `BrokerTypeDef` object passed to the function has been deleted. Either the client with which it was associated was disconnected or destroyed, or the cache was flushed.

AW_ERROR_NO_MEMORY

Ran out of memory while trying to complete the requested operation

AW_ERROR_NO_PERMISSION

You do not have the necessary permission to do the operation you attempted. For example:

- Attempting to write to a read-only envelope field.
- Using a client group that does not contain your identity with the correct permissions when reconnecting or creating a new `BrokerClient`.
- Attempting to publish an event type that is not allowed by your `BrokerClient` object's client group.

For more information on client group permissions and can publish permissions see *Administering webMethods Broker*.

AW_ERROR_NOT_IMPLEMENTED

The requested function call is not implemented.

AW_ERROR_NULL_PARAM

A null value was passed for a parameter that requires a value.

AW_ERROR_OUT_OF_RANGE

A parameter value is outside the accepted range. For example, getting a sequence subset using negative indexes will return this error.

AW_ERROR_PROTOCOL

Internal failure communicating with the Broker.

AW_ERROR_SECURITY

A security problem was encountered that prevented the operation from being completed.

AW_ERROR_SUBSCRIPTION_EXISTS

You attempted to create a new subscription with the `awNewSubscription` function, using an event type and filter that has already been used for another subscription.

AW_ERROR_TIMEOUT

This error is returned by functions such as `awGetEvent` when an event is not received within the specified time-out interval.

AW_ERROR_UNKNOWN_BROKER_NAME

The Broker whose name was specified on a call to the `awNewBrokerClient` or `awReconnectBrokerClient` function was not found.

AW_ERROR_UNKNOWN_CLIENT_GROUP

The client group specified on a call to the `awNewBrokerClient` or `awReconnectBrokerClient` function was not found.

AW_ERROR_UNKNOWN_CLIENT_ID

The client ID specified on a call to the `awReconnectBrokerClient` was not found.

AW_ERROR_UNKNOWN_EVENT_TYPE

The specified event type was not found on the Broker. For example, calling the `awNewBrokerEvent` function on a non-existent type.

AW_ERROR_UNKNOWN_INFOSET

The specified info set was not found for the event type.

AW_ERROR_UNKNOWN_KEY

The platform key specified on a call to the `awGetPlatformInfo` function has no defined value.

AW_ERROR_UNKNOWN_NAME

The specified distinguished name does not exist in the appropriate certificate file.

B Parameter Naming Rules

■ Overview	408
■ Length Restriction	408
■ Restricted Characters	408
■ Reserved Words	409
■ System Parameters	409
■ Broker Parameter Restrictions	409

Overview

This appendix describes the rules for naming webMethods Broker system parameters, including host names, distinguished names, passwords, Broker names, and client group names as well as event type names and event field names.

Note:

Unicode character values described in this section are represented as `\u####`.

Length Restriction

Broker uses a network data representation that requires all 2-byte unicode characters to be converted to 6 byte ANSI strings. Since the maximum parameter length is 255 bytes, this means that a parameter containing only Unicode characters may not be any longer than 42 bytes. Each of the following Broker parameters must have a length of either 1 to 255 ANSI characters or 1 to 42 Unicode characters:

- Broker name
- Client group
- Client ID
- Event type name
- Event field name
- Infoset name
- Infoset field name
- Territory name

Restricted Characters

With a few restrictions, a Broker parameter may be specified using any Unicode or ANSI characters. This allows you to use a variety of languages when naming items such as a Broker or event type. However, some characters are restricted and cannot be used.

- All non-printable ANSI characters (defined as the two ranges `\u0000` to `\u001F` and `\u007F` to `\u009F`).
- The ANSI characters '@', '/', and '\'.

Note:

In addition to these restricted characters, specific types of Broker parameters may place further restrictions on allowable characters. See the following section for complete details.

Reserved Words

any	boolean	byte	char
const	date	double	enum
false	final	float	int
long	null	short	string
struct	true	typedef	unicode_char
unicode_string	union	unsigned	

EventType and InfoSet Names

The name of an event type or infoSet may not be any of the words shown in [“Reserved Words” on page 409](#) or the table below. The entire event type or infoSet name is checked against these two lists of reserved words. This means that you cannot use the name "broker" for an event type, but you may use the name "my::broker".

Additional reserved words for event type names and infoSet names

acl	broker	client	clientgroup
event	eventtype	extends	host
import	infoSet	server	

System Parameters

The following table shows common restrictions on system parameters, which depend on your specific platform.

System Parameter	Restrictions
Broker Host name	Limited by most systems to printable 7-bit ASCII.
File names and Passwords	Limited by most systems to 8-bit ANSI characters.
Distinguished Names	Limited to printable 7-bit ASCII characters.

Broker Parameter Restrictions

The following table shows the restrictions placed on various Broker parameters.

Broker Parameter	Restrictions
Territory name Broker name Client Group name Client Id	Cannot begin with a “#” or contain any of the restricted characters described on “Restricted Characters” on page 408 .
Application name Platform Info Key	Cannot contain any of the restricted characters described on “Restricted Characters” on page 408 .
Event Type name Event Field name InfoSet name InfoSet Field name	Cannot begin with a digit (0-9) or with an underscore. May only contain alphanumeric characters, underscores, dollar symbols ('\$'), and Unicode characters greater than \u009F. May not contain symbols, white space, and non-printable ANSI characters.
Platform Info value	No restrictions.
Filter strings	No restrictions, other than the syntax restrictions described in “Managing Event Types” on page 113 .
Format strings	No restrictions, but the '\$', '{', and '}' characters are used as part of the format syntax.

C Unicode String Functions

■ Overview	412
■ Basic Unicode String Functions	412
■ Unicode String Conversion Functions	415
■ Unicode String Copy/Conversion Functions	417
■ Unicode String Length Functions	420

Overview

This appendix describes the string manipulation functions provided by the webMethods Broker C API for platforms that do not provide built-in support for Unicode strings. The functions are grouped into the following four categories:

- Basic Unicode String Functions
- Unicode String Conversion Functions
- Unicode String Copy/Conversion Functions
- Unicode String Length Functions

Note:

Many of the functions described in this appendix accept string parameters in *ANSI* or *UTF-8* format. The ANSI string format consists of 8-bit, ISO-Latin-1 characters. UTF-8 is the *Unicode Transform Function 8* format, which allows each Unicode character in a string to be represented as a sequence of up to four 8-bit, ISO-Latin-1 characters.

Basic Unicode String Functions

The following are the basic Unicode string functions:

- [awStrcpyUC](#)
- [awStrncpyUC](#)
- [awStrdupUC](#)
- [awStrlenUC](#)
- [awStrcmpUC](#)
- [awStrncmpUC](#)
- [awStrcatUC](#)
- [awStrchrUC](#)
- [awStrstrUC](#)

awStrcpyUC

```
charUC* awStrcpyUC(  
    charUC *dest,  
    charUC *src);
```

dest The string being copied to.

scr The Unicode string being copied.

Copies the *src* Unicode string, including the '\0' termination character, to the *dest* Unicode string and returns a pointer to the new string.

awStrncpyUC

```
charUC* awStrncpyUC(  
    charUC *dest,  
    charUC *src,  
    size_t n);
```

dest The string being copied to.

src The Unicode string being copied.

n The number of characters to be copied.

Copies no more than *n* Unicode characters into *dest* and returns a pointer to the *dest* string.

If *src* is less than *n* characters long, '\0' characters will be added to *dest*. If *src* is more than *n* characters long, *dest* will not be terminated with a '\0' character.

awStrdupUC

```
charUC* awStrdupUC(  
    charUC *st);
```

st The Unicode string being copied.

Creates a duplicate of the Unicode string *st* and returns a pointer to the newly created string.

The caller is responsible for freeing the return value.

awStrlenUC

```
int awStrlenUC(  
    charUC *st);
```

st The string whose length is to be returned.

Returns the number of bytes in the string *st*.

awStrcmpUC

```
int awStrcmpUC(  
    charUC *st1,  
    charUC *st2);
```

st1 The string to be compared to *st2*.

st2 The string to be compared to *st1*.

Compares the Unicode strings *st1* and *st2* and returns one of the following values:

- -1 is returned if *st1* is less than *st2*.
- 0 is returned if *st1* is equal to *st2*.
- 1 is returned if *st1* is greater than *st2*.

awStrncmpUC

```
int awStrncmpUC(  
    charUC *st1,  
    charUC *st2,  
    size_t n);
```

st1 The string to be compared to *st2*.

st2 The string to be compared to *st1*.

n The number of characters to be compared.

Compares the first *n* characters of the Unicode strings *st1* and *st2* and returns one of the following values:

- -1 is returned if *st1* is less than *st2*.
- 0 is returned if *st1* is equal to *st2*.
- 1 is returned if *st1* is greater than *st2*.

awStrcatUC

```
charUC *awStrcatUC(  
    charUC *dest,  
    charUC *src);
```

dest The Unicode string to which *src* is to be concatenated.

src The Unicode string to be concatenated onto *dest*.

Concatenates the Unicode string *src*, including the '\0' termination character, to the string *dest* and returns a pointer to *dest*.

awStrchrUC

```
charUC *awStrchrUC(  
    charUC *st,  
    charUC c);
```

st The Unicode string to be searched.

c The Unicode string to be concatenated onto *dest*.

Returns a pointer to the first occurrence of the Unicode character *c* in the string *st*. A NULL value is returned if *c* is not found in the string.

awStrstrUC

```
charUC* awStrstrUC(
    charUC *st1,
    charUC *st2)
```

st1 The Unicode string to be searched.

st2 The Unicode string to be concatenated onto *dest*.

Locates the first occurrence of the null-terminated Unicode string *st2* in the null-terminated Unicode string *st1*.

Returns a pointer to *st1* if *st2* is empty. Returns a NULL pointer if *st2* does not occur in *st1*. In all other cases, returns a pointer to the first character of the first occurrence of *st2* within *st1*.

Unicode String Conversion Functions

The following are the Unicode string conversion functions:

- [awAtoUC](#)
- [awAtoUTF8](#)
- [awUCtoA](#)
- [awUCtoUTF8](#)
- [awUTF8toA](#)
- [awUTF8toUC](#)

awAtoUC

```
charUC* awAtoUC(
    char *a_st);
```

a_st The ANSI string to duplicated.

Creates a duplicate of the ANSI string *a_st*, converting the duplicate to a Unicode string in the process. A pointer to the newly created Unicode string is returned.

Any UTF-8 representations, in the form `\u####`, contained in the ANSI string will be converted to the appropriate Unicode character in the duplicate string.

The caller is responsible for freeing the return value.

awAtoUTF8

```
char* awAtoUTF8(  
    char *a_st);
```

a_st The ANSI string to duplicated.

Creates a duplicate of the ANSI string *a_st*, converting the duplicate to UTF-8 format in the process. A pointer to the newly created UTF-8 string is returned.

Any occurrences in the ANSI string of the form `\u####` will be considered to be UTF-8 representations.

The caller is responsible for freeing the return value.

awUCtoA

```
char *awUCtoA(  
    charUC *uc_st);
```

uc_st The Unicode string to duplicated.

Creates a duplicate of the Unicode string *uc_st*, converting the duplicate to an ANSI string in the process. A pointer to the newly created ANSI string is returned.

Unicode characters are represented in the resulting ANSI string as an escape sequence, such as `\uf138`.

The caller is responsible for freeing the return value.

awUCtoUTF8

```
char* awUCtoUTF8(  
    charUC *uc_st);
```

Creates a duplicate of the Unicode string *uc_st*, converting the duplicate to UTF-8 format in the process. A pointer to the newly created UTF-8 string is returned.

The caller is responsible for freeing the return value.

awUTF8toA

```
char* awUTF8toA(  
    char *utf8_st);
```

Creates a duplicate of the UTF-8 string *uc_st*, converting the duplicate to an ANSI string in the process. A pointer to the newly created ANSI string is returned.

Unicode characters are represented in the resulting ANSI string as an escape sequence, such as `\uf138`.

The caller is responsible for freeing the return value.

awUTF8toUC

```
charUC* awUTF8toUC(
    char *utf8_st);
```

Creates a duplicate of the UTF-8 string *uc_st*, converting the duplicate to a Unicode string in the process. A pointer to the newly created UTF-8 string is returned.

The caller is responsible for freeing the return value.

Unicode String Copy/Conversion Functions

The following are the Unicode string copy/conversion functions:

- [awCopyAtoUC](#)
- [awCopyAtoUTF8](#)
- [awCopyUCtoA](#)
- [awCopyUCtoUTF8](#)
- [awCopyUTF8toA](#)
- [awCopyUTF8toUC](#)

awCopyAtoUC

```
int awCopyAtoUC(
    charUC *uc_dest,
    char *a_src,
    size_t n);
```

uc_dest The Unicode copy of the ANSI string.

a_src The ANSI string to be copied.

n The maximum number of 16-bit characters that may be contained in *uc_dest*.

Copies an ANSI string, *a_src*, to a Unicode string, *uc_dest*. The parameter *n* represents the maximum number of 16-bit characters that may be set in *uc_dest*. The destination string will not be null-terminated if *n* is equal to the size of *uc_dest*.

Any UTF-8 representations, in the form `\u####`, contained in the ANSI string will be converted to the appropriate Unicode character in the duplicate string.

Returns the number of 16-bit characters used in *uc_dest*.

awCopyAtoUTF8

```
int awCopyAtoUTF8(  
    char *utf8_dest,  
    char *a_src,  
    size_t n);
```

<i>utf8_dest</i>	The UTF-8 copy of the ANSI string.
<i>a_src</i>	The ANSI string to be copied.
<i>n</i>	The maximum number of characters that may be contained in <i>utf8_dest</i> .

Copies an ANSI string, *a_src*, to a UTF-8 string, *utf8_dest*. The parameter *n* represents the maximum number of bytes that may be set in *utf8_dest*. The destination string will not be null-terminated if *n* is equal to the size of *utf8_dest*.

Any occurrences in the ANSI string of the form `\u####` will be considered to be UTF-8 representations.

Returns the number of bytes used in *utf8_dest*.

awCopyUCtoA

```
int awCopyUCtoA(  
    char *a_dest,  
    charUC *uc_src,  
    size_t n);
```

<i>a_dest</i>	The ANSI copy of the Unicode string.
<i>uc_src</i>	The Unicode string to be copied.
<i>n</i>	The maximum number of characters that may be contained in <i>a_dest</i> .

Copies a Unicode string, *uc_src*, to an ANSI string, *a_dest*. The parameter *n* represents the maximum number of bytes that may be set in *a_dest*. The destination string will not be null-terminated if *n* is equal to the size of *a_dest*.

Unicode characters are represented in the resulting ANSI string as an escape sequence, such as `\uf138`.

Returns the number of bytes used in *a_dest*.

awCopyUCtoUTF8

```
int awCopyUCtoUTF8(
    char *utf8_dest,
    charUC *uc_src,
    size_t n);
```

utf8_dest The UTF-8 copy of the ANSI string.

uc_src The Unicode string to be copied.

n The maximum number of characters that may be contained in *utf8_dest*.

Copies a Unicode string, *uc_src*, to a UTF-8 string, *utf8_dest*. The parameter *n* represents the maximum number of bytes that may be set in *utf8_dest*. The destination string will not be null-terminated if *n* is equal to the size of *utf8_dest*.

Returns the number of bytes used in *utf8_dest*.

awCopyUTF8toA

```
int awCopyUTF8toA(
    char *a_dest,
    char *utf8_src,
    size_t n);
```

a_dest The ANSI copy of the UTF-8 string.

utf8_src The UTF-8 string to be copied.

n The maximum number of characters that may be contained in *a_dest*.

Copies a UTF-8 string, *utf8_src*, to an ANSI string, *a_dest*. The parameter *n* represents the maximum number of bytes that may be set in *a_dest*. The destination string will not be null-terminated if *n* is equal to the size of *a_dest*.

Unicode characters are represented in the resulting ANSI string as an escape sequence, such as `\uf138`.

Returns the number of bytes used in *a_dest*.

awCopyUTF8toUC

```
int awCopyUTF8toUC(
    charUC *uc_dest,
    char *utf8_src,
    size_t n);
```

uc_dest The Unicode copy of the UTF-8 string.

utf8_src The UTF-8 string to be copied.

n The maximum number of characters that may be contained in *uc_dest*.

Copies a UTF-8 string, *utf8_src*, to a Unicode string, *uc_dest*. The parameter *n* represents the maximum number of bytes that may be set in *uc_dest*. The destination string will not be null-terminated if *n* is equal to the size of *uc_dest*.

Returns the number of bytes used in *uc_dest*.

Unicode String Length Functions

The following are the Unicode string length functions:

- [awAlengthInUC](#)
- [awAlengthInUTF8](#)
- [awUClengthInA](#)
- [awUClengthInUTF8](#)
- [awUTF8lengthInA](#)
- [awUTF8lengthInUC](#)

awAlengthInUC

```
int awAlengthInUC(  
    char *a_st);
```

a_st The ANSI string whose length is to be returned.

Returns the length, in 16-bit character values, of an ANSI string represented as a Unicode string.

awAlengthInUTF8

```
int awAlengthInUTF8(  
    char *a_st);
```

a_st The ANSI string whose length is to be returned.

Returns the length, in bytes, of an ANSI string represented as a UTF-8 format string.

awUClengthInA

```
int awUClengthInA(  
    charUC *uc_st);
```

uc_st The Unicode string whose length is to be returned.

Returns the length, in characters, of a Unicode string represented as an ANSI string.

awUClengthInUTF8

```
int awUClengthInUTF8(  
    charUC *uc_st);
```

uc_st The Unicode string whose length is to be returned.

Returns the length, in bytes, of a Unicode string represented as a UTF-8 format string.

awUTF8lengthInA

```
int awUTF8lengthInA(  
    char *utf8_st);
```

utf8_st The UTF-8 string whose length is to be returned.

Returns the length, in characters, of a UTF-8 string represented as an ANSI string.

awUTF8lengthInUC

```
int awUTF8lengthInUC(  
    char *utf8_st);
```

utf8_st The UTF-8 string whose length is to be returned.

Returns the length, in 16-bit character values, of a UTF-8 string represented as a Unicode string.

D Transactional Client Processing with Adapters

■ Overview	424
■ Transaction Processing	424
■ Using Transaction Processing	428

Overview

This chapter describes the webMethods Broker C API for implementing applications that publish events using a transaction processing model. Reading this chapter will help you to understand:

- The various levels of transaction support offered by adapters
- How to create a transaction identifier
- How to begin and end a transaction
- How to perform a COMMIT operation
- How to perform a ROLLBACK operation
- How to perform a SAVEPOINT operation

Transaction Processing

Transaction processing allows your Broker client to group the events it publishes as a single unit of work called a transaction. A transaction either completes successfully, is rolled back to some known earlier state, or it fails. Once all of the events that make up a transaction have been published, your Broker client then ends the transaction. Depending on the capabilities of the adapter your Broker client is using, a transaction can be ended by committing the transaction, setting a save point, or rolling back to the beginning of the transaction.

Transaction Levels

The webMethods Broker system allows adapters to offer several levels of transaction processing. An adapter may not support transaction processing at all, or it may provide one of following levels of support:

- Pseudo-transaction support
- Basic transaction support
- Conversational transaction support

Note:

To determine the transaction level supported by the adapter you are using, consult the documentation for that adapter.

Non-transactional Behavior

Adapters that do not support any level of transaction processing have the following characteristics:

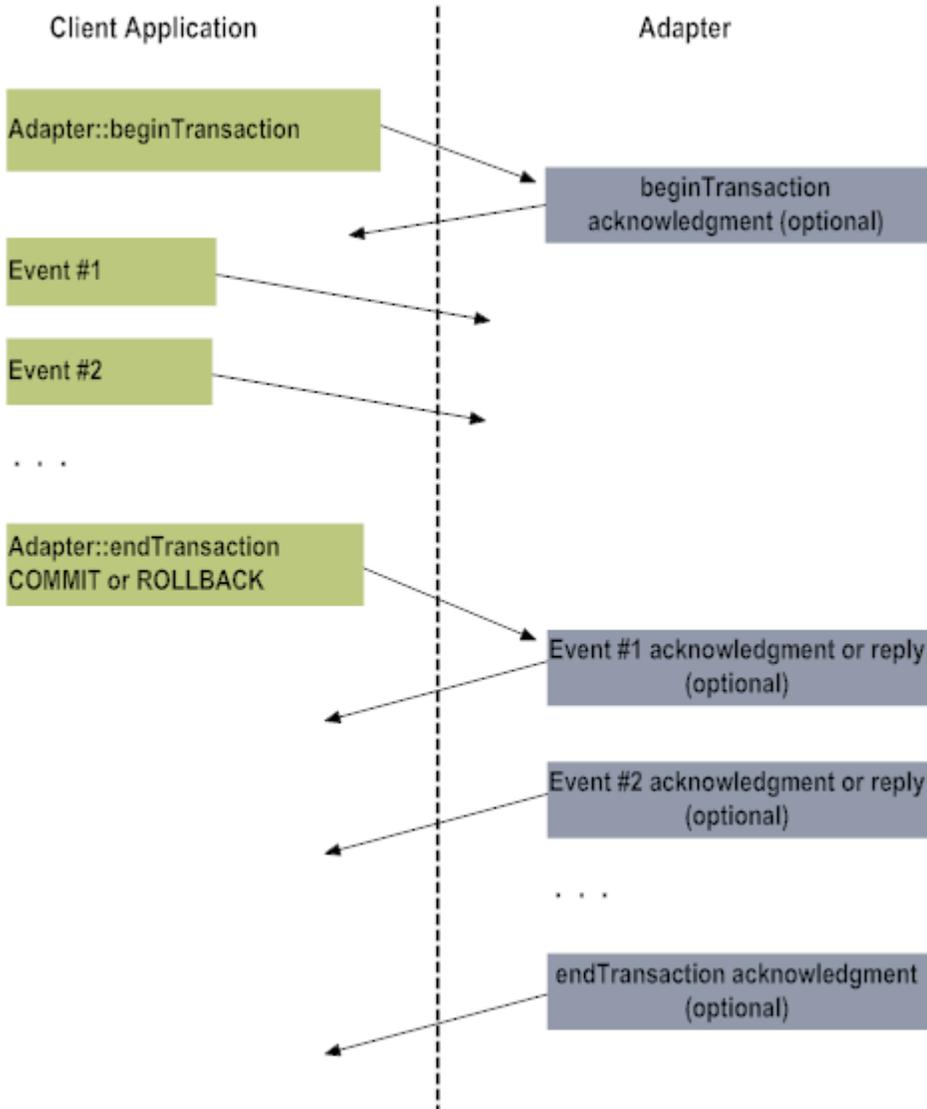
- BEGIN and END operations are ignored and no acknowledgements are sent for either operation.
- Events are processed as they are received and the ROLLBACK operation is not supported.
- An error with one or more events does not prevent the processing of subsequent events.

Pseudo-transaction Support

This level of support allows your Broker client to reliably rollback to the beginning of the transaction. The adapter processes the transaction's events in a tight sequence, making it less likely that your data will become inconsistent, but it does not offers protection in the event of a failure. Here are the other features of pseudo-transaction support:

- The `SAVEPOINT` operation is not supported.
- After a `ROLLBACK` operation, all request events published so far are discarded and will not generate any reply events, not even error replies. The `ROLLBACK` operation returns an acknowledgment.
- The `COMMIT` operation simply returns an acknowledgment.
- Any request event that generates an error after a `COMMIT` operation will cause all subsequent events in the same transaction to be discarded without being processed. Furthermore, any changes prior to the error *will not* be undone and ending the transaction will generate an error.
- Depending on the design of the adapter, a `COMMIT` or `ROLLBACK` operation can fail even if all the previous request events succeeded, so you should always request an acknowledgment when ending a transaction.

Pseudo-transaction support

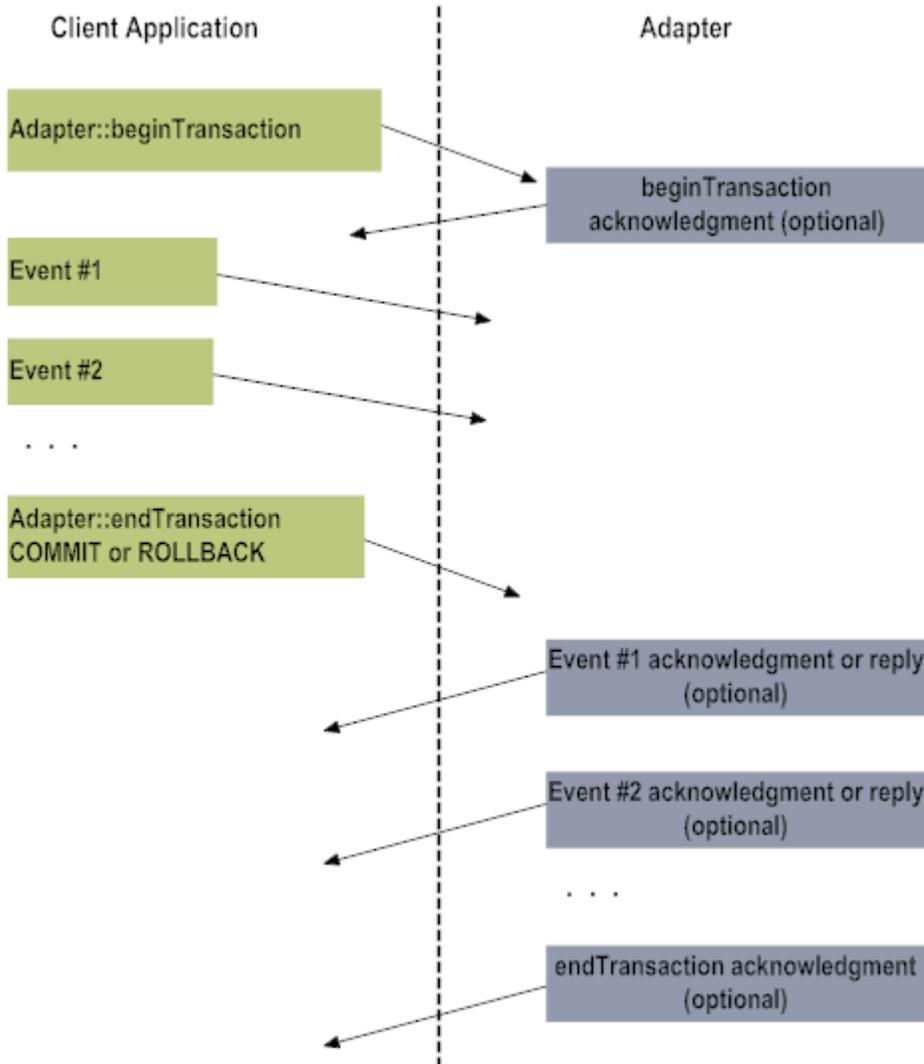


Basic Transaction Support

An adapter with basic transaction support allows your Broker client to begin a transaction, publish the events that make up the transaction, and then COMMIT or ROLLBACK the events. The SAVEPOINT operation is not supported. Here are the other features of basic transaction support:

- After a ROLLBACK operation, all request events in the transaction are discarded and will not generate any reply events, not even error replies.
- The COMMIT and ROLLBACK operations both return an acknowledgment.
- Any request event that generates an error following a COMMIT operation will cause subsequent events to be discarded. Furthermore, any changes from prior events will be rolled back. If an error occurs, the Adapter::endTransaction event will also return an error.

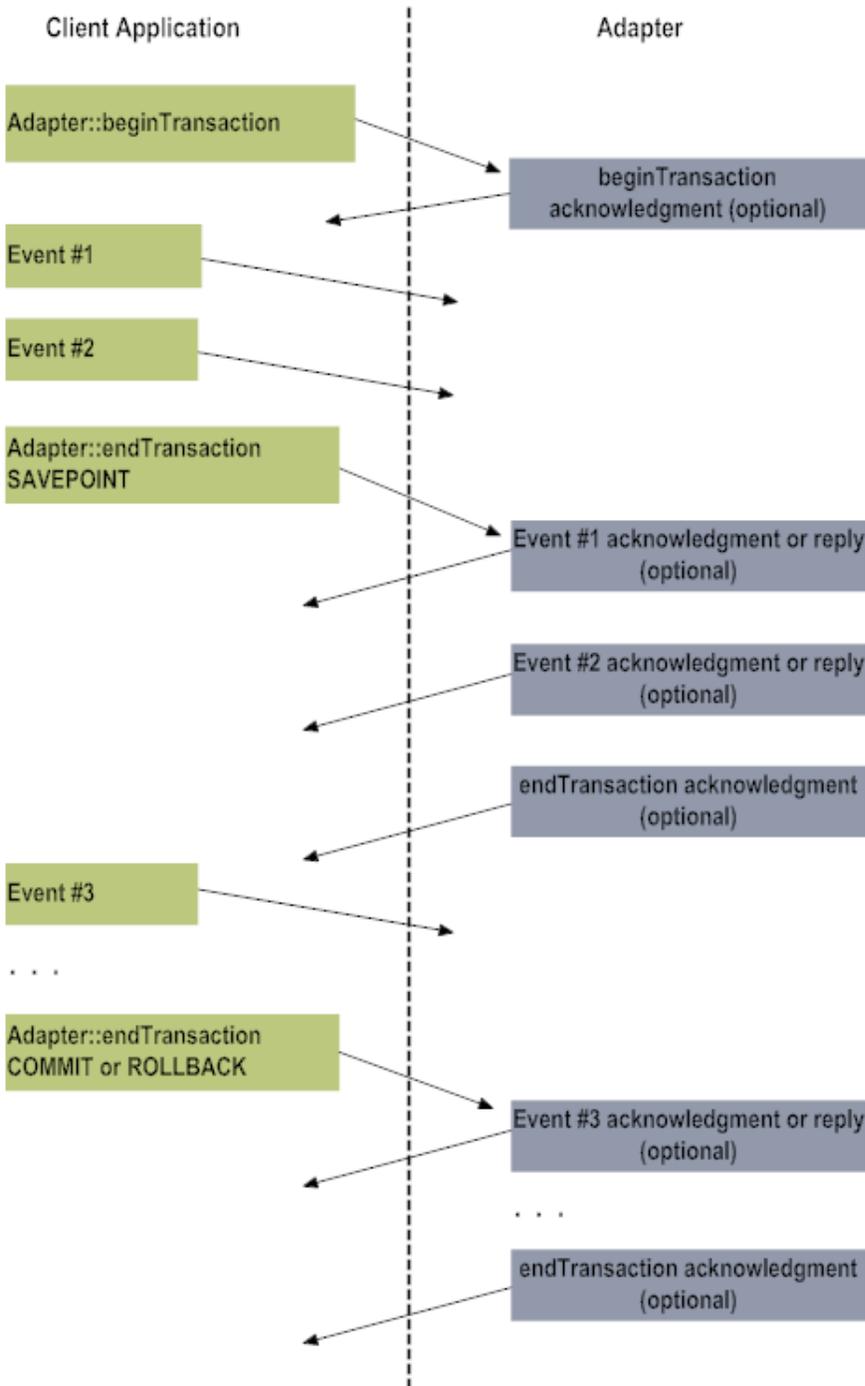
Basic transaction processing



Conversational Transaction Support

Conversational transaction support provides all of the features of basic transaction support and adds support for the `SAVEPOINT` operation. Where other adapters return an error for the `SAVEPOINT` operation, this type of adapter processes queued events in the transaction, then leaves the transaction open for future `SAVEPOINT`, `COMMIT`, or `ROLLBACK` operations.

Conversational transaction processing



Using Transaction Processing

To use transaction processing, your application should follow these steps:

1. Obtain a transaction identifier that will be used on all published events that are part of the transaction.

2. Begin the transaction.
3. Publish the events that make up the transaction.
4. End the transaction, which may consist of a COMMIT, ROLLBACK, or SAVEPOINT operation.

Obtaining a Transaction ID

Use the [awMakeTxTransactionId](#) to obtain a unique transaction identifier for your Broker client. You must set the `transactionId` envelope field for each event that you want to publish as part of the transaction.

The following example illustrates how to invoke the [awMakeTxTransactionId](#) function

```
BrokerClient c;
BrokerError err;
char** transId;
. . .
// Obtain a transaction id.
err = awMakeTransactionId( c, transId);
if (err != AW_NO_ERROR) {
    printf("Error creating transaction ID\n%s\n",
        awErrorToString(err));
    return 0;
}
. . .
```

Beginning a Transaction

Use the [awBeginTransaction](#) method to begin the transaction. This function publishes an `Adapter::beginTransaction` event, so your Broker client must have permission to publish this event. The function requires the following parameters:

- The transaction ID.
- The desired transaction level.
- A list of client identifiers for all the Broker clients that are allowed to interact with events in the transaction. This list can be `NULL` if you want all subscribers to participate in the transaction.
- An area where a tag field can be returned. If your Broker client does not wish to receive an acknowledgment event for the `Adapter::beginTransaction` event that is to about be published, this parameter should set to `NULL`.

Example of invoking the [awBeginTransaction](#) function

```
BrokerClient c;
BrokerError err;
char **transId;
int transTag;
. . .
// Obtain a transaction id.
. . .
```

```
// Begin the transaction
err = awBeginTransaction( c, transId, AW_TRANSACTION_LEVEL_BASIC,
    0, NULL, &transTag);
if (err != AW_NO_ERROR) {
    printf("Error beginning transaction\n%s\n",
        awErrorToString(err));
    return 0;
}
. . .
```

Transaction Level Negotiation

When beginning a transaction, your client specifies the level of transaction processing it desires from the adapter. The adapter will return an error if the requested level of transaction support is lower than the level it supports. The transaction levels are specified as one of the values shown below.

Name	Description
AW_TRANSACTION_LEVEL_ANY	The lowest level of transaction support. Used by the awBeginTransaction function to request any of the following levels of transaction support from an adapter.
AW_TRANSACTION_LEVEL_PSEUDO	The next highest level of transaction support, described on “Transaction Levels” on page 424 .
AW_TRANSACTION_LEVEL_BASIC	Described on “Basic Transaction Support” on page 426 .
AW_TRANSACTION_LEVEL_CONVERSATIONAL	The highest level of transaction support, described on “Conversational Transaction Support” on page 427 .

The adapter will return an error reply if the transaction level it offers is lower than the level you have requested.

Note:

The error reply will not be sent until the adapter receives the first request event.

Multiple Participants

Your client application can specify the client identifiers of all adapters that are allowed to participate in the transaction at the time the [awBeginTransaction](#) function is invoked. If you do not wish to restrict the adapters that can participate in the transaction, you may specify a NULL list. If the participant list is not NULL, only those adapters appearing in the list should process events in the transaction and all other adapters should ignore the events.

Requesting Acknowledgment

If the `reply_tag` parameter is not `NULL`, an acknowledgment will be sent for the `Adapter::beginTransaction` event. The tag field will be set in `reply_tag` so that your application can correlate the acknowledgment event you receive with the `Adapter::beginTransaction` event that was published.

Note:

The acknowledgment reply will not be sent until the adapter receives the first request event.

Publishing Events within a Transaction

Your application publishes the events that make up the transaction in the manner described in [“Publishing and Delivering Events” on page 69](#).

Remember to set the `transactionId` envelope field of each event with the transaction identifier you used when you began the transaction.

Note:

Any event that is published with a `transactionId` that is not known to the adapter will return an error event.

Whether or not your application will receive an acknowledgment event or a reply event for each event it publishes depends on how the event type was defined.

Note:

Even if an adapter can successfully receive the `Adapter::beginTransaction` and `Adapter::endTransaction` events, it must have permission to subscribe to the request events that make up the transaction.

Ending a Transaction

Ending a transaction can involve a `COMMIT`, `ROLLBACK`, or `SAVEPOINT` operation. Use the [awEndTransaction](#) method to end the transaction. This function requires the following parameters:

- The transaction identifier.
- The mode, which should be either `TRANSACTION_MODE_COMMIT`, `TRANSACTION_MODE_SAVEPOINT`, or `TRANSACTION_MODE_ROLLBACK`.
- An area where a tag field can be returned. If your Broker client does not wish to receive an acknowledgment event for the `Adapter::endTransaction` event that is to about be published, this parameter should set to `NULL`.

Note:

Requesting a `SAVEPOINT` operation will return an error event if the adapter's transaction level does not support save points, but the transaction will remain open.

If the `reply_tag` parameter is not `NULL`, an acknowledgment will be sent for the `Adapter::endTransaction` event. The tag field will be set in `reply_tag` so that your application can correlate the acknowledgment event you receive with the `Adapter::endTransaction` event that was published.

The following example illustrates how to invoke the function:

```
BrokerClient c;  
BrokerError err;  
char **transId;  
int transTag;  
.  
.  
.  
    // Obtain a transaction id.  
    .  
    .  
    // end the transaction  
    err = awEndTransaction( c, transId, TRANSACTION_MODE_COMMIT,  
        &transTag);  
    if (err != AW_NO_ERROR) {  
        printf("Error ending transaction\n%s\n",  
            awErrorToString(err));  
        return 0;  
    }  
    .  
    .  
    .
```

Time-outs

A transaction can time-out if the adapter does not receive a `Adapter::endTransaction` event within a certain amount of time after the receipt of a `Adapter::beginTransaction` event. If a transaction times out, the adapter will rollback the transaction, causing an error to be returned from the first request event (not the `Adapter::beginTransaction` event).

If an adapter is shut down before the time-out interval expires and then is restarted, it may reset the time-out interval.

The time-out interval is defined when the adapter is configured. For more information, refer to the documentation for the particular adapter you are using.